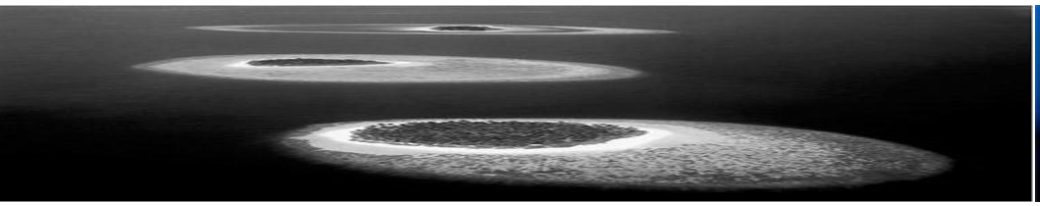
Proteus Configurator v.2.1 User Guide



Revision 1.01

Author: Erik Danielsson

January 2013

promineo

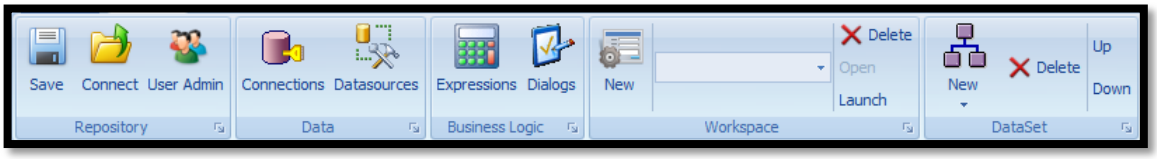
Contents

Chapter 1: Introduction 4
Chapter 2: Getting started 5
Chapter 3: Setting up Groups and Users 6
Active Directory Security Model 6
Proteus' Security Model 6
Chapter 4: Creating a new Workspace 9
Typical Steps: 9
Optional Advanced Steps: 10
General: 10
General Options: 10
Gantt Settings: 10
Periodization Engine Settings: 10
Setting up Default Report Header/Footer and Print Setup: 11
Admin Rights to Workspace Features: 12
Advanced – Time Units: 12
Defining the Cutoff Rule 13
Advanced - Value Units: 13
Chapter 5: Setting up DataConnections 14
Chapter 6: Creating DataSources 15
Typical Steps: 15
Optional Advanced Steps: 16
How to make an SQL statement dynamic? 18
Chapter 7: Setting up DataSet(s) and their internal relationships 19
General 20
Typical Steps: 20
Advanced Settings: 21
Using a DataSet as Task Link definitions for another (parent) DataSet: 23
Gantt Bars 24
Gantt Markers 25
Some examples of using Gantt Markers: 26
Calculated Columns 27
Dynamic Calculated Columns 27
The {{Global Expression}} syntax 28
Static Calculated Columns 29
Fieldnames with special meaning 29
Testing a calculated column expression 30
Inherited Fields 31
Periodized Fields 32
Generating rows in histogram data from formula 33
Bypassing the Periodization Engine with pre-periodized data 34
Pre-Periodized Data 35
Completion Matrices 36
Creating a "Continuous Coloring Scheme" 39
Group Summaries 40
Shadow Data 41
Shadow Type: “Snapshot” 41
Advanced custom formatting options for compare deviations: 42
Shadow Type: “Advanced” 43
How to load more than one historical set of data from snapshot table. 44
Alert Settings 45
Union Data 46
Column Values 46
Custom Exports 47
Example of generating an Export template: 49
Example of setting shape properties for an existing Export template: 51

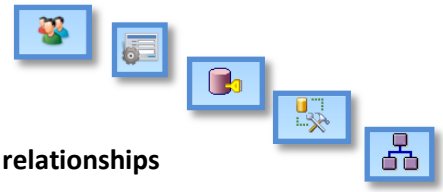
Chapter 8: Creating a dynamic interface – The Proteus DataLayout 52
Some useful hints on working with Graphs in a DataLayout: 55
Controlling Chart details using special calculated column names: 57
Example of how to make an interactive custom Profile Editor for Safran: 57
Some useful hints on defining Gauge properties: 59
Working with unbound controls in a DataLayout 61
Chapter 9: Defining Expression 63
Some help with built-in examples from Example Picker 65
Some examples of non-trivial Expressions 66
Chapter 10: Defining custom Dialogs 67
Chapter 11: Setting up Action Buttons 69
Chapter 12: Creating Help information for a workspace 71
Chapter 13: Adding visual information using Annotations 73
How to get annotations in a DataLayout Graph 76
Chapter 14: Appendix A – DataSource formats for Calendars 77
Chapter 15: Appendix A2 – DataSource formats for Profiles 79
Chapter 16: Appendix A3 – DataSource for Custom ColorMapping 81
Chapter 17: Appendix A4 – DataSource for Annotations 82
Chapter 18: Appendix B: How to modify source data from Proteus 83
Chapter 19: Appendix C: Syntaxes for assigning colors 85
Chapter 20: Appendix C2: Syntax for controlling Style in Grid Cells 87
Chapter 21: Appendix D: Internally generated fields 88
Chapter 22: Appendix E: Working with a TreeList control 90
Setting up a TreeList 90
Using a TreeList in a Custom Dialog 90
Set rules for drag & drop of nodes in TreeList 91
Chapter 23: Appendix F: Markup syntax to format text 92
Chapter 24: Appendix G: Exporting data to Safran Planner 93
Chapter 25: Appendix H: Working with Project Folders 95
Chapter 26: Appendix I: Working with the Overlay Control 96
Chapter 27: Appendix Ib: The Overlay Control's polygon format 99
Chapter 28: Appendix J: Folder Structure and Security 100
Chapter 29: Appendix K: Formatting syntax for Time Units 103
Copyright Notice

Chapter 1: Introduction

The idea behind Proteus is to wire up interface(s) to any existing (hierarchical) data, and provide some useful tools for viewing/interacting with the data in various ways: Time-phased, Gantt Charted, Pivoted, Custom Reports, etc.
This document will give an insight into how this is accomplished, using the Proteus Configurator application.



The chapter outline is as follows:
***Note**: You can have a "Hello World" Workspace up and running when the typical steps up to **7a** are completed (shown in bold below). This can be done in a matter of minutes, so the threshold for getting started does not need to be very high.*



2. **Getting Started**
3. **Setting up Groups and Users** (User Admin)
4. **Creating a new Workspace**
5. **Setting up DataConnection(s)**
6. **Creating DataSources.**
7. **Setting up DataSet(s) and their hierarchical relationships**
   a. **General**
   b. Gantt Bars
   c. Gantt Markers
   d. Calculated Columns
   e. Inherited Fields
   f. Periodized Fields
   g. Pre-Periodized Data
   h. Completion Matrices
   i. Group Summaries
   j. Shadow Data
   k. Alert Settings
   l. Union Data
   m. Column Values
   n. Custom Exports

8. Creating a dynamic interface – The Proteus DataLayout
9. Defining Expressions
10. Defining custom Dialogs
11. Setting up Action Buttons
12. Creating Help Information for a Workspace
13. Adding visual information using Annotations

Chapter 2: Getting started

Proteus Configurator is an application that needs to be installed locally on your machine. The purpose of the application is to build so called "Workspace(s)" to be run from Proteus. The Workspaces configuration files are stored in a folder called the Proteus Repository Folder.
This chapter will get you started.

Deploying Proteus Configurator:
The software is installed through the technology called “click-once", where a Url (http://clickonce.promineo.no/ProteusConfigurator/2.1/) is used to deploy the software, as well as ensure that the latest version is running on the local machine.
Note that you must use Proteus version 2.0 to "run" a Workspace created with this version of Proteus Configurator.

What is a Repository folder?:
A Repository is a folder on a file-share that contains the Workspace configurations as well as the various user settings related to a Workspace (public and private layouts, filters, etc).
If no repository folder has been set up before, it must be created manually. An example of such a path could be:
P:\Proteus\Repositories\MyRepositoryName

One Repository can contain a number of workspaces. All Workspaces will have access to the same Data Connections, Data Sources, Dialogs, etc. inside the same Repository. If you would like to create an independent (set of) workspace(s), you should consider creating a separate Repository for this purpose. Please review your license agreement to verify how many repositories your organization is allowed to operate.

In Appendix, you can read more about the subfolders that may be found in a repository, as well as a discussion on access security.

Sample Repository with various Workspaces :
Upon request (proteus@promineo.no or from your regional Proteus vendor) you can be given access to a Repository called "**Proteus Demo Repository**", containing several examples that are referenced in this document.

Chapter 3: Setting up Groups and Users

For each repository it is possible to create Groups and assign Users to these. The Group membership also controls which rights one has regarding Workspace access as well as access to features within a Workspace (such as: who can save public layouts at runtime, etc).

There are two distinct modes of security to use when setting up role based access to Proteus:

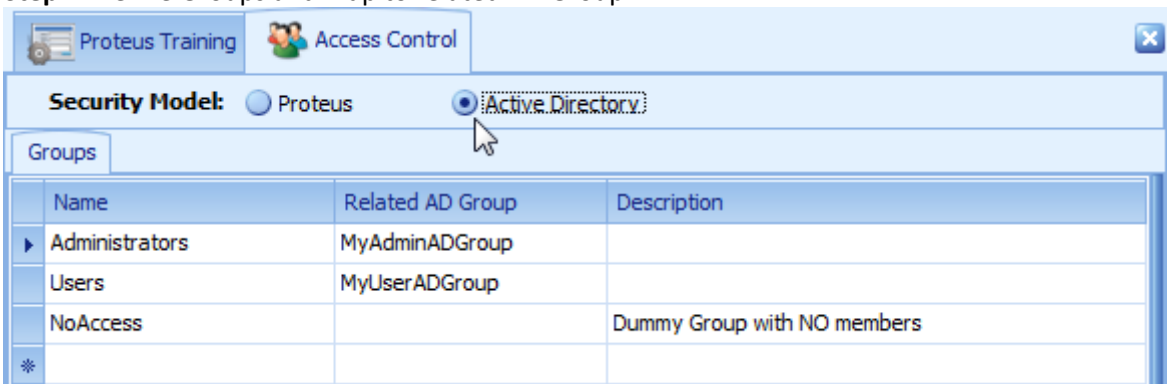
1. Active Directory Security Model.
2. Proteus' Security Model

In either of these models you need to define Groups to be used to internally when assigning "access rights" to Workspaces and features within a Workspace. Please note that the names of your groups can be arbitrary, but it is always useful if they carry some meaning to their actual function.

Active Directory Security Model

This model requires less work and is therefore the natural choice - unless Active Directory, for whatever reason, is not an option.

Step 1: Define Groups and map to related AD Group:



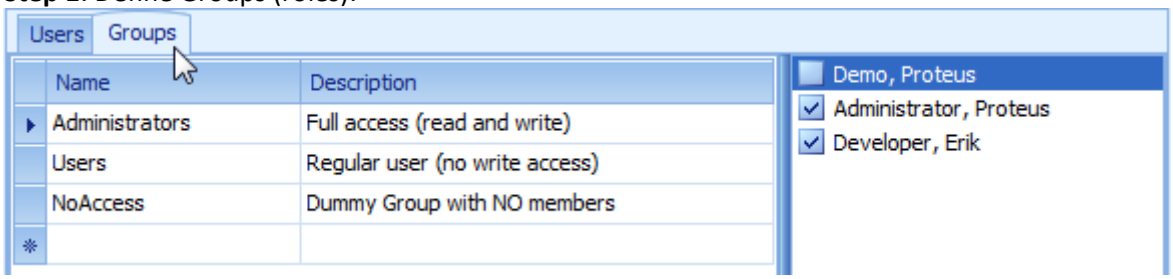
The only thing that is required in Active Directory Model is to map an AD Group to a Proteus Group.

Note: In this mode there is no possibility to create "Users" as seen in "Proteus' Security Model" below, but only map an AD Group to an internal Proteus Group.

Proteus' Security Model

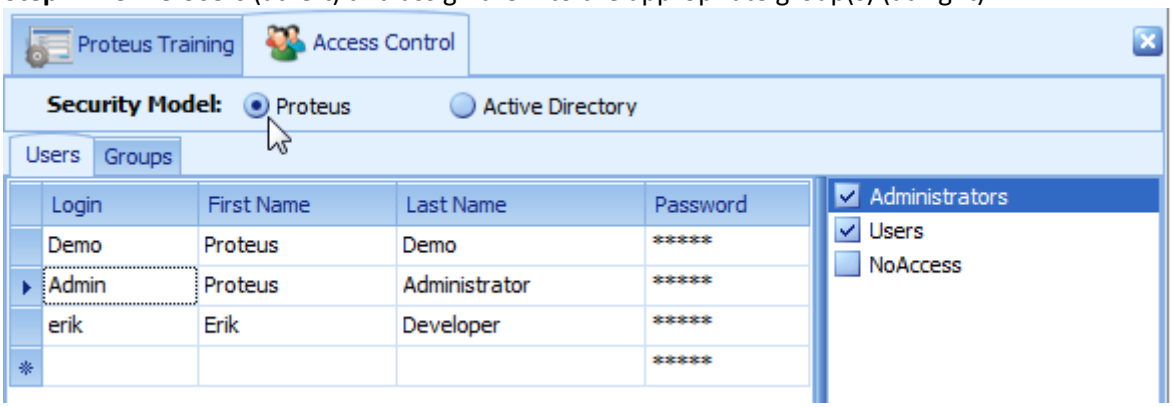
This is the model to use if Active Directory is not used in your organization, or other technical reasons exist why it is not be available.

Step 1: Define Groups (roles):



Note that the list at above right will be blank the first time, when no users have been defined.

Step 2: Define Users (at left) and assign them to the appropriate group(s) (at right):



There are no limits to how many Groups and Users can be created, and these are stored in the file called “Security.proteussecurity” (having passwords encrypted so they are not readable if file is opened in notepad).

Note:
Please note that the users do not get the opportunity to change their passwords when logging on to Proteus (or Configurator). As mentioned in “Notes on Security” in this document, Proteus assumes that there are sufficient security barriers outside the application, so it is therefore no need to be overly protective of these passwords. They can be made visible by double clicking on the Password column header – and returned invisible by double click it again.

Scenarios where shared project Login may be considered
Note: The following is only relevant for Proteus Security Model

Let's imagine the following scenario:
You have three sets of project specific data for these Projects, stored in the same table:

1. Project_Cat
2. Project_Dog
3. Project_Horse

One of the columns in your datasource (having fieldname "ProjectName") contains the string from one of the above project names (i.e "Cat", "Dog", "Horse").
Each of the projects has around 50 people wanting to access their data, so it would be quite cumbersome to create 150 individual Proteus users for this purpose.

Suggested solution:
Instead you could create **3** users, and assign them to a read only Group:

1. Cat_User
2. Dog_User
3. Horse_User

Furthermore, if you wanted to have admin users for each project, you could create 3 more users, and assign them to an Admin Group (of your choice):

1. Cat_Admin
2. Dog_Admin
3. Horse_Admin

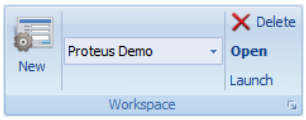
You could then (through techniques described later in this document – how to make a SQL statement dynamic) let the "Where Clause" of the SQL that loads data be affected by the Login name (after the "_User" or "_Admin" part of the name has been removed), resulting in a SQL like this (for a "Cat" user):

```
Select * from MyProjectDataTable where ProjectName = 'Project_Cat'
```

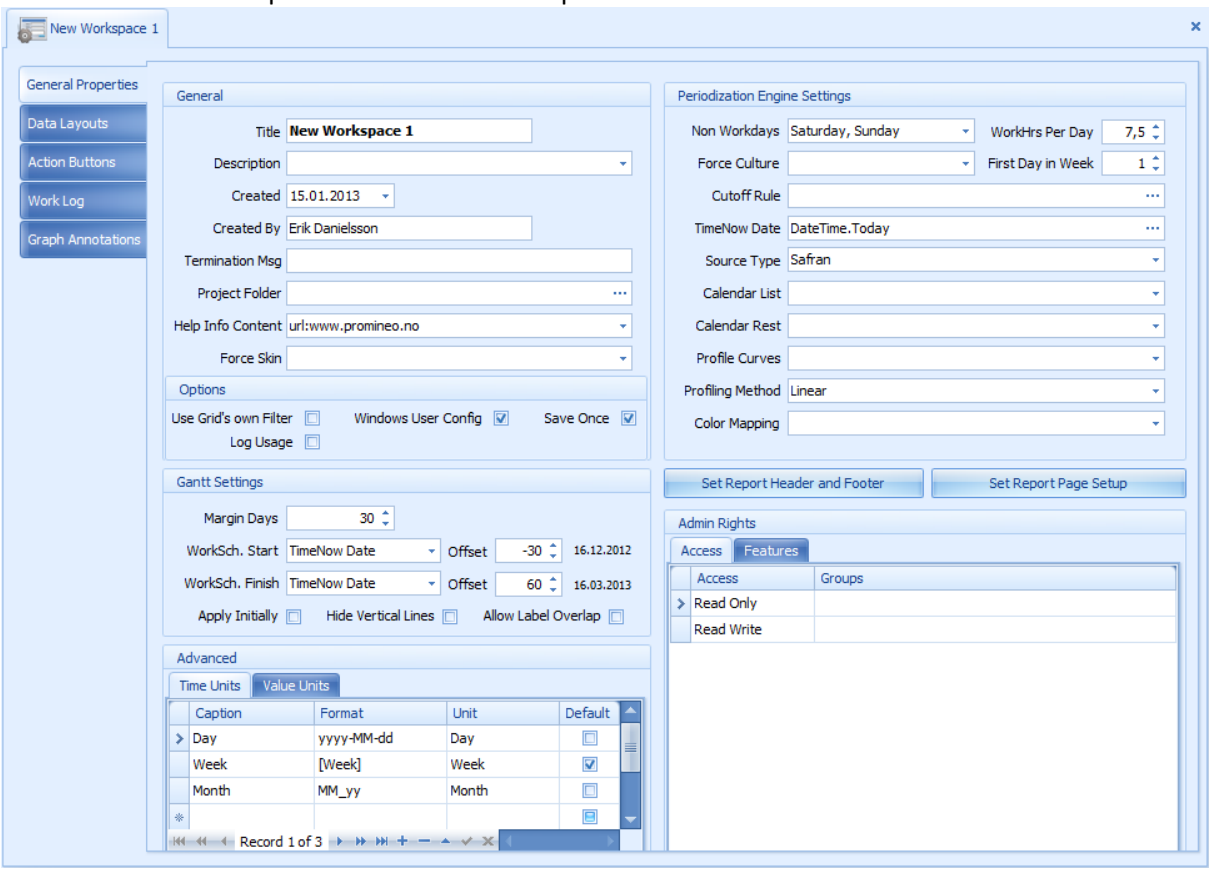
Note that in an "**Active Directory Security Model**", the challenge would be solved using a similar manipulation of SQL, but using the Group name instead of the User name to produce the "Where Clause".

Chapter 4: Creating a new Workspace

A Repository can contain multiple Workspaces, and this section describes how a new one is created, and the various settings it has.



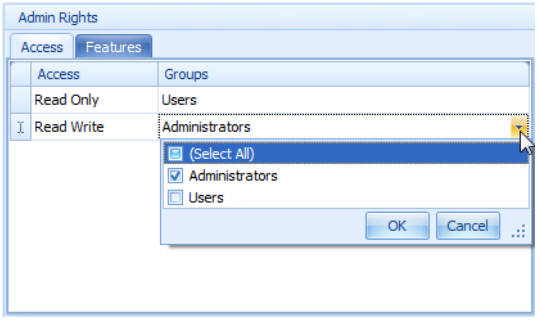
Press the "New Workspace" icon on the Workspace toolbar.



Screenshot above shows the form for setting General Properties for a Workspace

Typical Steps:

1. **Title**: Fill in a title for the Workspace (change "New Workspace 1" to your choice)
2. **Description**: Define a descriptive text to be shown when Workspace is selected in login dialog. It gives the user an indication of what the Workspace does. Line feed can be inserted in text by typing "<CR>".
3. **Give access to Workspace**:



In the "Admin Rights" section, under the "Access" tab, you can control which Group(s) should have **Read Only** and which should have **Read Write** access.
"**Read Only**" implies that the Workspace can be opened in Proteus, but not modified in Configurator – this ability is given to the Group(s) assigned to "**Read Write**".

Optional Advanced Steps:

General:

4. **Termination Msg**: The text to show when workspace is closing, if any.
5. **Project Folder**: An expression, if any, that controls which folder to use for user settings. The evaluated result must match an existing folder name (created manually). In this way various sets of user settings can exist for different user groups/projects. See Appendix for more info.
6. **Help Info Content**: The link to html file showing help content for Workspace. Prefix with "url:" in this case, or "doc:" if you wish to refer to an existing document (.doc or .docx file). More..
7. **Force Skin**: The name (if any) of skin to use for this Workspace.

General Options:

8. **Use Grid's own Filter**: If checked, the grid(s) showing the loaded data will have an extra row at top acting as filter criteria entry. It this is not checked, a separate filter control, above the grid, will contain a customizable filter (that is saved together with the grid layout).
9. **Windows User Config**: Check if you wish to have Proteus create private folders (for saving user settings such as filters, layouts, etc) based on Windows Login Name (default), or leave it unchecked to use the Proteus Login Name for this purpose. The latter option allows for Private Settings to be shared between same Proteus Login.
10. **Save Once**: If checked, only one confirmation needs to be made to save all your changes. If unchecked, each DataSet will need separate confirmation before saving changes to it.
11. **Log Usage**: If checked, a log file will be updated each time a user logs on, to reflect the usage of each workspace, and to allow IT administrators to monitor activity, and identify who the users are (should this be necessary). Please note that this feature requires write access for all users on the Repository root folder, where the log file resides (UserLog.txt)

Gantt Settings:

12. **Margin Days**: The number of days to add before and after the loaded data's date range. This allows for visual "margins" in the Gantt Chart.
13. **Work Schedule Start/Finish**: Defines the time-range to use as the "WorkSchedule". The Gantt Chart as well as the Histogram has a button that can quickly select this time-range.
14. **Apply Initially:** Apply the above settings to Gantt Chart's time-range at startup.
15. **Hide Vertical Lines:** Hide the 3 vertical lines showing WorkSchedule start/finish and TimeNow.
16. **Allow Label Overlap:** Check if you want to allow Gantt Chart labels to overlap.

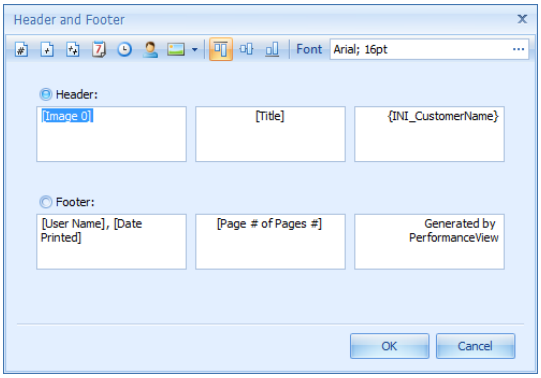
Periodization Engine Settings:

17. **Non Workdays**: Define which days of the week are working days for *Proteus' Default Calendar*. None of these days will get time-phased data.
18. **WorkHours Per Day**: This setting, together with "Non Workdays" above, define *Proteus' Default Calendar*. This Default Calendar will be used if no custom calendar is set up (see step 23 below), or if the calendar ID is not found in the loaded set of calendars. The "WorkHours Per Day" setting is important when converting hours to men.
19. **Force Culture**: Specify which culture (language) to use to when creating month names (in accordance with the formatting that is specified). Leave blank to use control panel settings.
20. **First Day in Week**: Specify which day is 1st day of work-week. 1=Monday, 7= Sunday.
21. **TimeNow Date**: Supply an Expression here that defines "Today's date". It controls where the "TimeNow" line is shown in Gantt Chart, as well as affecting the periodization engine's way of time-phasing for certain spread modes (Start to TimeNow and TimeNow to Finish). Press ellipsis at right to see the evaluated date.
    Typical examples:
    a. Today's date: `DateTime.Today`
    b. One Week Ago: `DateTime.Today.AddDays(-7)`

c. Today's week's first date (Monday): `Calendar.GetFirstCurrentDate("w", DateTime.Today)`
d. Today's month's first date: `Calendar.GetFirstCurrentDate("m", DateTime.Today)`
e. Fixed date (Jan. 30, 2013): `[2013-01-30]`
f. Lookup date from table in database: `cn#MyConnection.ExecuteScalar("Select MyDate From MyTable Where MyValue=99")`
g. Get date from custom dialog: `dlg#MyDialogID#MyDateControlID`

22. **Source Type**: Allows specifying which type of planning tool to get calendar data from. Currently supported formats are: **Safran** and **Primavera** (P6).
23. **Calendar List/Rest**: Specify which pre-defined DataSources to use as calendar definitions: List (with name of calendar and workhours per day) and Rest (not relevant for Primavera Calendar Type since this is part of the List definition) – controls which days are considered non-working days. If these are given, the settings for Proteus' Default Calendar (see 17 and 18 above) will not be used, unless there is a reference to a calendar that is not loaded.
The expected format for these DataSources can be seen in Appendix.
Note: The calendars are inherited down from parent to child DataSets.
24. **Profiling Method**: Any of these: Linear (Default), Bezier, Step. More here…
25. **Color Mapping**: Specify which pre-defined DataSource to use for split histogram coloring schemes. More here…

Setting up Default Report Header/Footer and Print Setup:

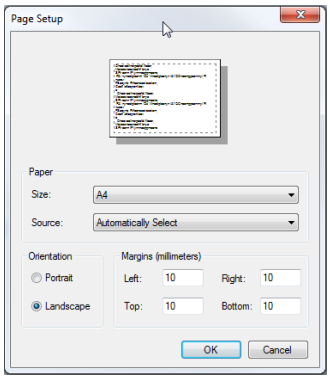


By clicking one of these two buttons (2 pages back), these 2 dialog forms will appear:

Header and Footer:

As seen at left, there are three "boxes" for header, and three for footer, were the font properties can be set for each of these groups.
It is possible to pick images to any of these boxes, and this is done by selecting (clearing) the box's content, and picking an image from list. The list of images and their naming convention is described in Appendix.
Note that it is possible to guide values generated by Expressions to any of these by referring to it in curly brackets (as shown in example "{INI_CustomerName}").
In addition, there are a few predefined "markers" that can be used as placeholder for content that are passed to report at runtime:

Marker	Description
[Title]	The title defined for each individual report.
[User Name]	The windows name of current user.
[Date Printed]	The date of print (i.e "today's date").
[Filter]	The filter definition when report was created.
[Page # of Pages #]	The paging info. Can be used like this as well "Page **[Page #]** of **[Pages #]**"



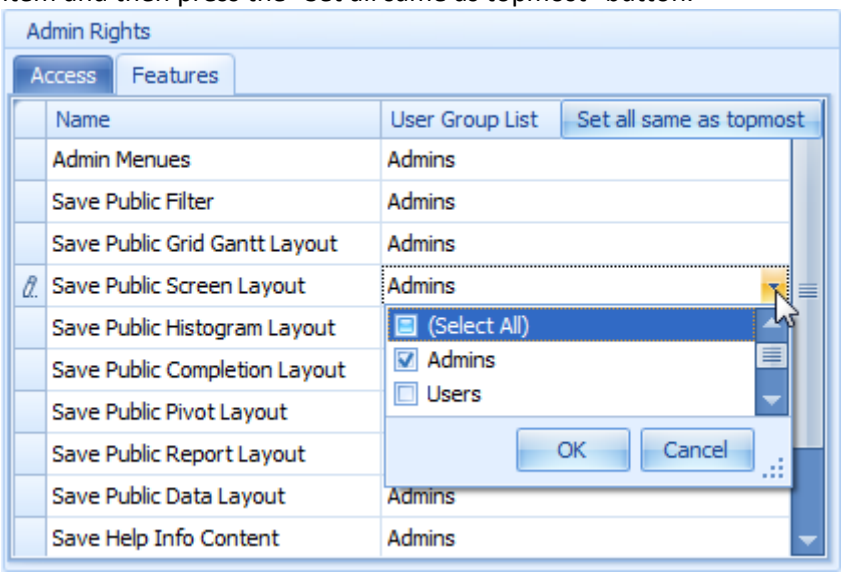
Page Setup:

As seen at left, it is possible to set the default values for:
Paper size, Orientation, Margins

These are used for reports generated "ad hoc", i.e without going through the Report Wizard. However, settings here are also used by the Wizard as its default settings, to be overridden for each individual report.

Admin Rights to Workspace Features:

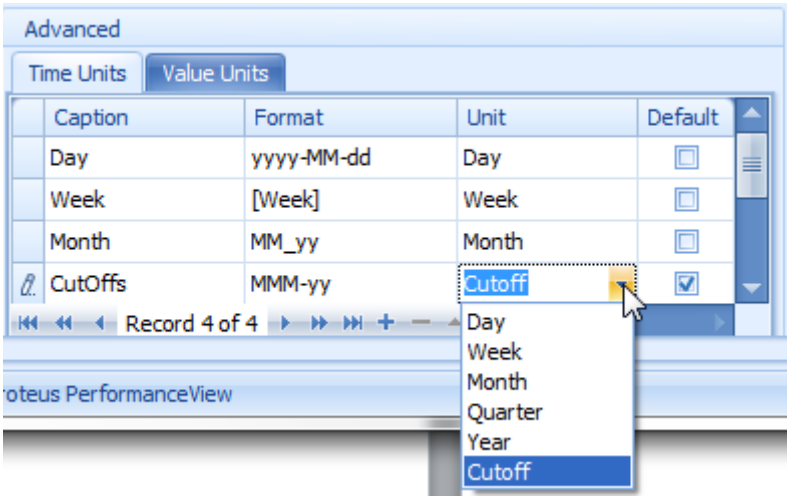
Controls which Group(s) can save the various Public Layouts, as well as see the Admin Menu items. Screenshot below shows the complete list of "Features", in the Name column. For each, a list of Groups can be selected that should have access to the feature. The quick way is to set the topmost item and then press the "Set all same as topmost" button.



The quick way is to set the topmost value, and then press the button to fill down.

Note: The tab called "Access" is described in "Typical Steps" under item 3.

Advanced – Time Units:

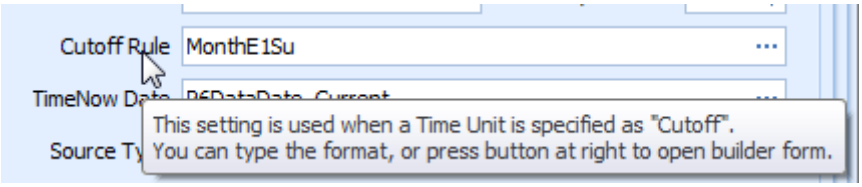


Controls which Time Units are available in the Histogram (time-phased result) and Completion Matrix. The one you want to use the first time you start Proteus should be checked as “Default”, but keep in mind that this setting will be overwritten by the Histogram/Completion Matrix layouts that are set as "Public Default".

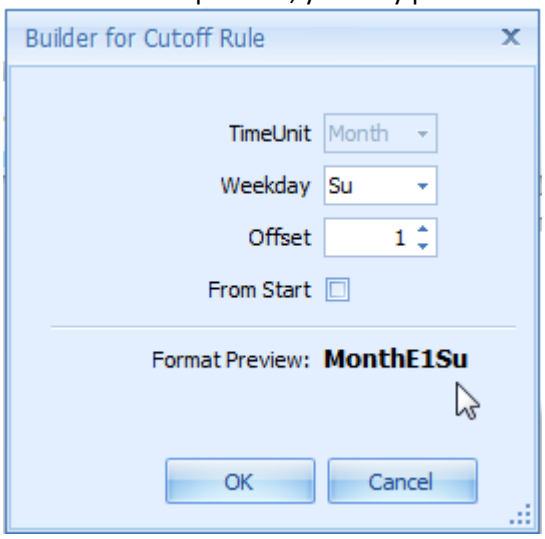
See Appendix for more info on formatting options of Time Units.

Defining the Cutoff Rule

When the Unit is set to "Cutoff", then the Format will still be used to control the visual presentation of the dates, *BUT* the *actual dates* that are used for the period can be controlled from the setting shown here:


Cutoff Rule
MonthE1Su
This setting is used when a Time Unit is specified as "Cutoff".
You can type the format, or press button at right to open builder form.

As seen in tooltip above, you may press the ellipsis to launch a "builder":



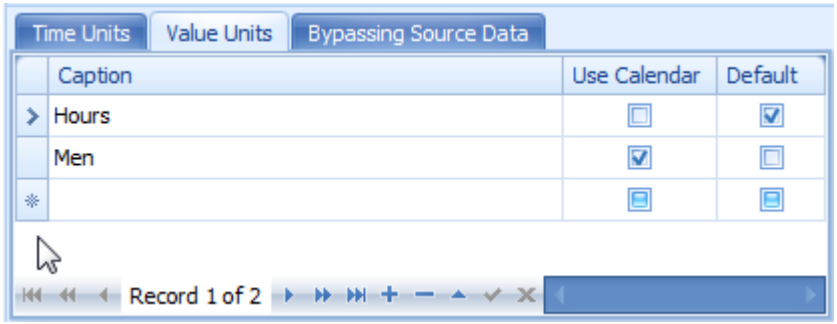
Syntax definition:
[Month<Ref><Cnt><Wd>]
<Ref> = **E** (End) or **S** [Start]
<Cnt> = **Count from Ref** (where 1 is last/first)
<Wd> = **WeekDay**: Mo, Tu, We, Th, Fr, Sa, Su

Example resulting in second to last Sunday in month becoming the end of each month's cutoff periods:
[**MonthE2Su**]

Advanced - Value Units:

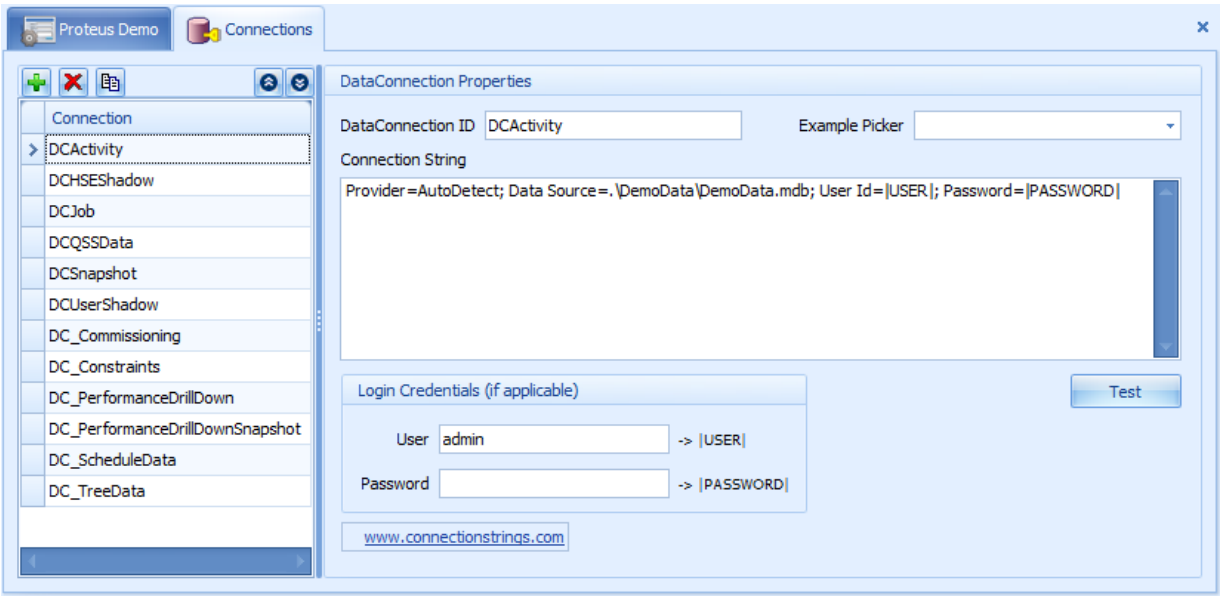
Allows for customizing which “value units” to show in the Histogram. The one to use the first time Proteus is started can be checked as “Default”. Again, remember that this setting will be overwritten by the Histogram/Completion Matrix layouts that are set as "Public Default".

If “Use Calendar” is checked, then the timephased quantity will be divided by calendar workhours per day. Typically this is used to convert hours to men.



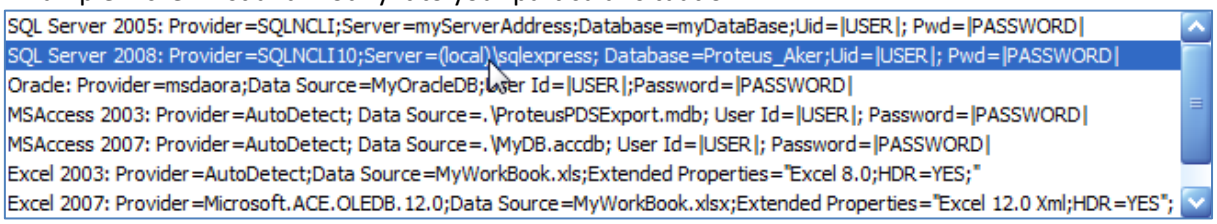
Chapter 5: Setting up DataConnections

A DataConnection implies a reference to a source of data. This can be any type of source that exposes data to the OLEDB standard. These include SQL Server, Oracle, Sybase, MySQL, MSAccess, Excel, Textfiles, Sharepoint Lists, and even HTML tables.



The form for defining Data Connections

Step 1: After giving the Connection an ID, create a connection string, by selecting one from the "Example Picker" list and modify it to your particular situation:


SQL Server 2005: Provider=SQLNCLI;Server=myServerAddress;Database=myDataBase;Uid=|USER|; Pwd=|PASSWORD|
SQL Server 2008: Provider=SQLNCLI10;Server=(local)\sqlexpress; Database=Proteus_Aker;Uid=|USER|; Pwd=|PASSWORD|
Oracle: Provider=msdaora;Data Source=MyOracleDB;User Id=|USER|;Password=|PASSWORD|
MSAccess 2003: Provider=AutoDetect; Data Source=.\ProteusPDSExport.mdb; User Id=|USER|; Password=|PASSWORD|
MSAccess 2007: Provider=AutoDetect; Data Source=.\MyDB.accdb; User Id=|USER|; Password=|PASSWORD|
Excel 2003: Provider=AutoDetect;Data Source=MyWorkBook.xls;Extended Properties="Excel 8.0;HDR=YES;"
Excel 2007: Provider=Microsoft.ACE.OLEDB.12.0;Data Source=MyWorkBook.xlsx;Extended Properties="Excel 12.0 Xml;HDR=YES";

Step 2: Add credentials (User and Password) in the "Login Credentials" boxes (if applicable).

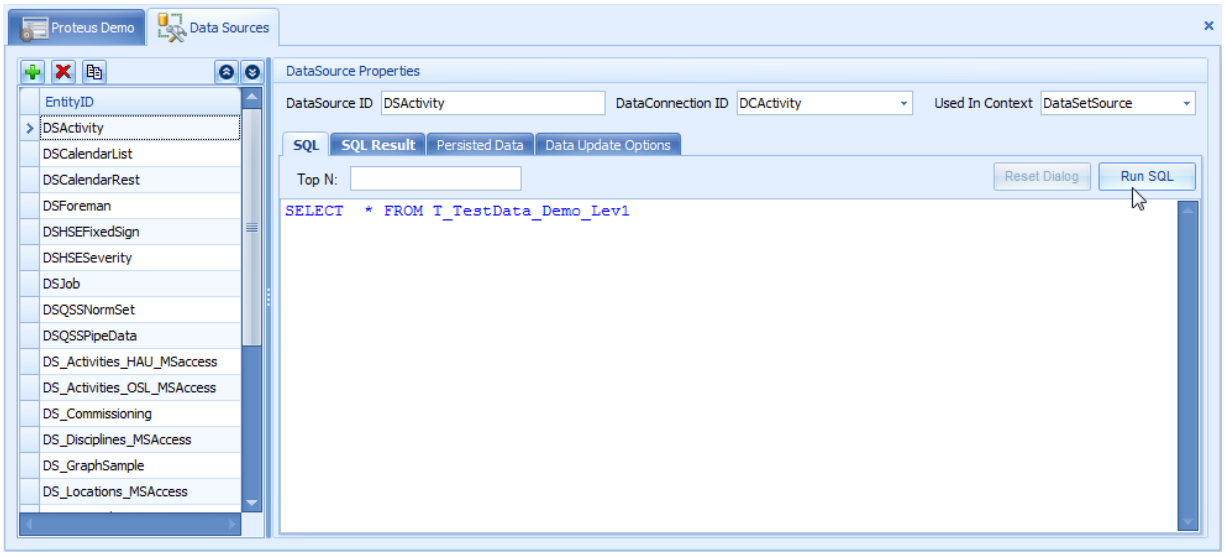
Step 3: **P**ress "Test"-button to see if it works.

Advanced info:
If the Example Picker does not cover your particular situation, you may press the link at lower left to go to a site containing virtually any example of an OLEDB connection string (www.connectionstrings.com). Please note that only the alternatives starting with the text "Provider=..." are applicable.

Comments on credentials: In order to avoid hard-coding the user name, and particularly the password into the readable "Connection String" textbox, it is possible to use special replacement markers "**|USER|**" and "**|PASSWORD|**" that will be replaced with the "Login Credentials" settings at bottom of form. In this way the password will not be directly readable (and it is saved encrypted for further security).

Chapter 6: Creating DataSources

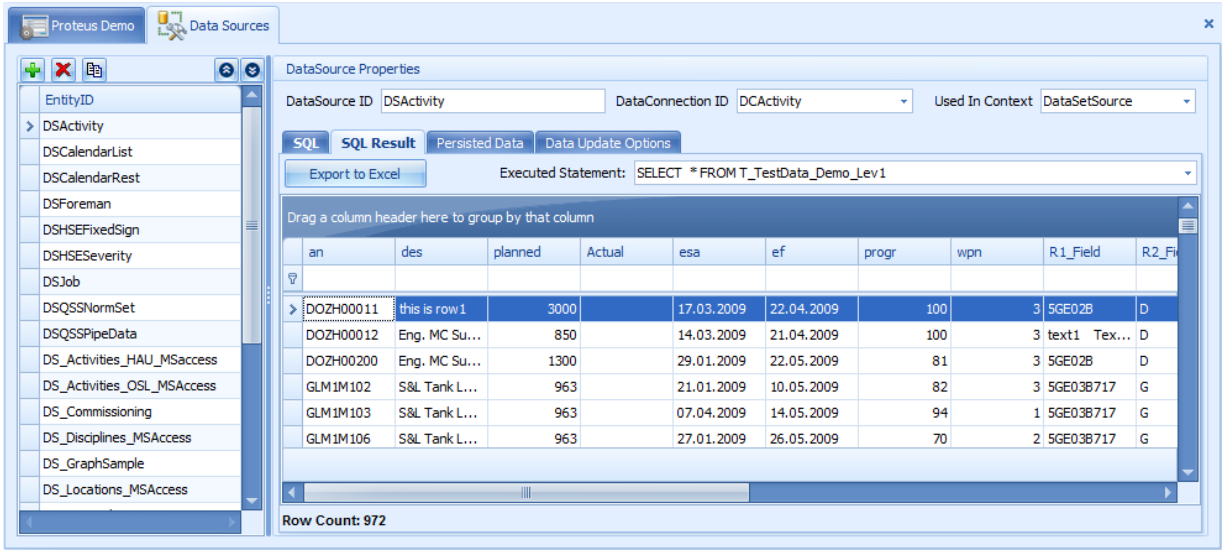
A DataSource is the specific selection of data from a DataConnection, represented by a query (SQL) statement. The resulting data is used by Proteus in a so called "DataSet", where additional calculated columns may be added to the ones given by query.



The form for entering DataSources

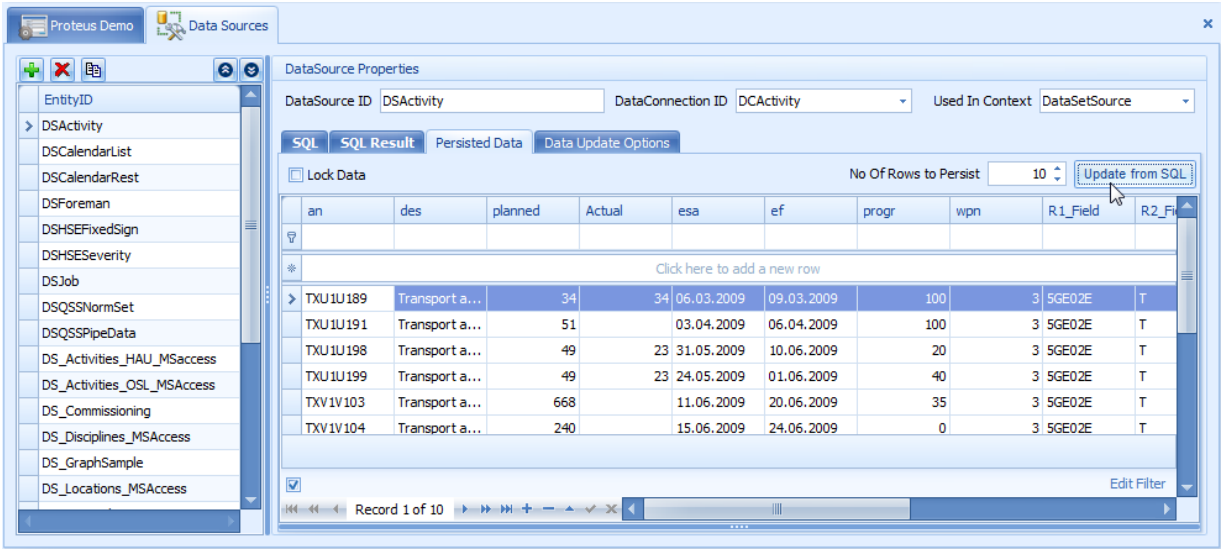
Typical Steps:

Step 1: Give the DataSource an ID (top left).
Step 2: Select which DataConnection to use from dropdown (top middle).
Step 3: Type the SQL statement for getting the data you want.
Step 4: Press "Run SQL" button to see result of query:


Proteus Demo
Data Sources
DataSource Properties
DataSource ID DSActivity
DataConnection ID DCActivity
Used In Context DataSetSource
SQL
SQL Result
Persisted Data
Data Update Options
Export to Excel
Executed Statement: SELECT * FROM T_TestData_Demo_Lev1
Drag a column header here to group by that column
an
des
planned
Actual
esa
ef
progr
wpn
R1_Field
DOZH00011 this is row 1 3000 17.03.2009 22.04.2009 100 3 5GE02B D
DOZH00012 Eng. MC Su... 850 14.03.2009 21.04.2009 100 3 text1 Tex... D
DOZH00200 Eng. MC Su... 1300 29.01.2009 22.05.2009 81 3 5GE02B D
GLM1M102 S&L Tank L... 963 21.01.2009 10.05.2009 82 3 5GE03B717 G
GLM1M103 S&L Tank L... 963 07.04.2009 14.05.2009 94 1 5GE03B717 G
GLM1M106 S&L Tank L... 963 27.01.2009 26.05.2009 70 2 5GE03B717 G
Row Count: 972

The Result of the SQL is shown in separate tab

Step 5: Go to "Persisted Data" – press the "Update from SQL" button.

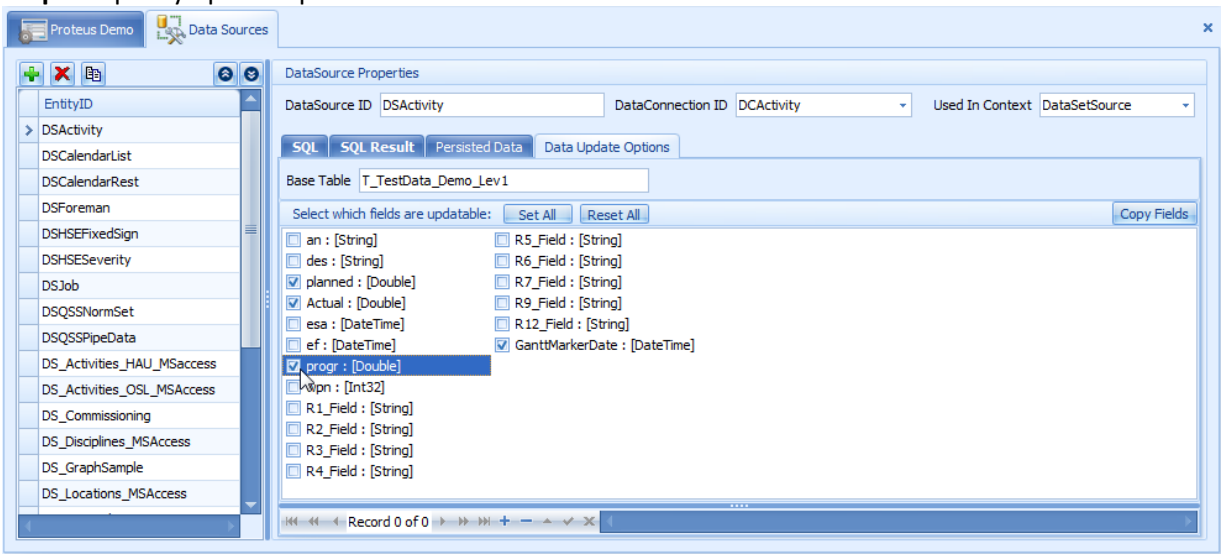


Tab where data can be persisted – to be used as sample data, and to store field information.

The reason you need to persist some sample data is to store the field types for later use (so it knows which fields are dates, numbers, text, etc without making a call to the underlying database). The sample data is also useful when testing the effect of a calculated column.

Optional Advanced Steps:

Step 6: Specify update option for the DataSource.

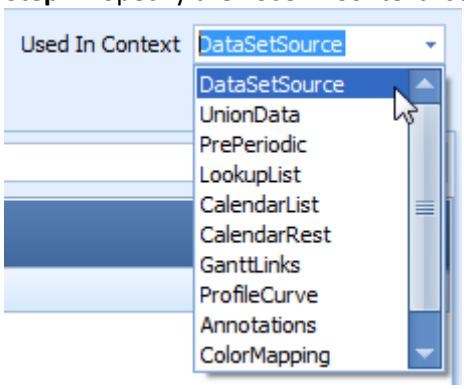


Tab where update options for DataSource can be specified.

The form shows a list of all the fieldnames, as well as their data types, for the DataSource (generated when pressing the "Persist" button described above). Here you can select which fields are updatable (for users with write access), by checking them off.

The "Base Table" setting is used when Proteus shall generate the Update, Insert and Delete SQL statement for modifying source data. More info on this in Appendix B: How source data can be modified from Proteus.

Step 7: Specify the "Use In Context" setting.



The choices for the "Used In Context" dropdown.

This allows classifying a DataSource according to its function. The default setting is "DataSetSource", and here is the explanation for each of the choices:

Setting	Description
DataSetSource	The DataSource will be used as a source for a DataSet (the default setting).
UnionData	The DataSource will be used to append data to a DataSet
PrePeriodic	The DataSource contains pre-periodized data to be used to bypass Proteus' periodization engine, and be shown directly in Histogram.
LookupList	The DataSource is used to populate various dropdowns in Grid columns or in a DataLayout.
CalendarList	The DataSource contains a list of all the calendars to be used for Proteus' periodization engine. In case the Workspace has been defined as having Source Type "Primavera", then the non-working days for the various calendars are defined in this DataSource as well (as an encoded field). More info on this is found in Appendix A.
CalendarRest	Works in tandem with the above type of DataSource, and is used in case the Source Type is "Safran", where a separate source is required to define the non-working days for each calendar. More info is found in Appendix A.
GanttLinks	The DataSource is used to provide relational link information between objects in a DataSet. More info on this is found here.
ProfileCurve	The DataSource contains the definitions of Profile Curves. The format of this DataSource depends on which Source Type is set. More info on this is given in Appendix A2.
Annotations	The DataSource contains information used to render annotation information (vertical lines, stripes, callouts) in any of the following settings: Histogram, Gantt Chart, Graph in DataLayout. More info on this is given in Appendix A4.
ColorMapping	The DataSource is used to define colors in Split Bar Histogram, for various split fields. More info on this is given in Appendix A3.
Undefined	An indication that this DataSource does not fall into any of the above categories, for whatever reason.

How to make an SQL statement dynamic?

It is sometimes useful to let the user select which data to load - especially if the number of rows in source is in the hundreds of thousands. It is possible to embed "expression placeholders" (names in curly brackets) in the SQL that are evaluated at runtime when Proteus loads its data. If such an Expression refers to a Dialog, *it* will be brought up so that the user can enter whatever value(s) it asks for.

Example of a dynamic SQL that brings up a custom dialog:

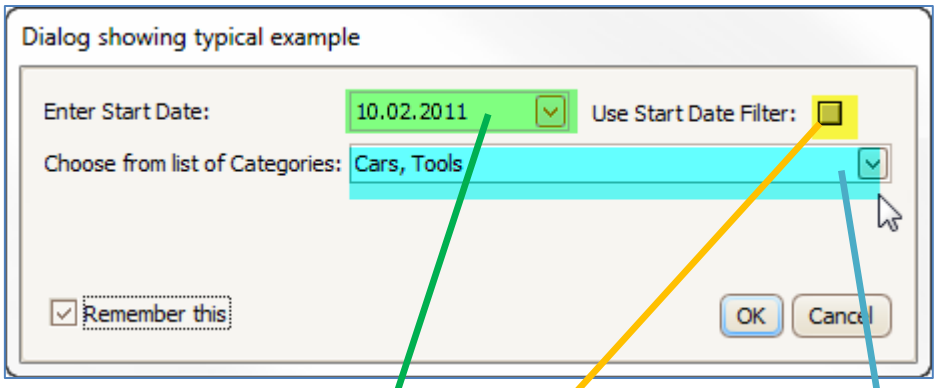
```
Select * from MyTable Where MyCategoryField = {SelectedCategoryValue}
```

The expression (defined under the Expression section) "SelectedCategoryValue" could then look something like this:

```
dlg#MyCategoryDialog#MyCategoryDropdown
```

Another (more sophisticated) example:

A somewhat more elaborate, but realistic, example would perhaps involve more than one field in the Where Clause of the SQL, and perhaps you wish to have a dialog like this to control how your SQL is built:



You have to create an expression called (e.g) "TypicalWhereClause" with this expression definition:

```
General.Iif
(
  dlg#TypicalExample#UseStartDateFilter,
  "MyDate >= '" + dlg#TypicalExample#MyDate.ToString("yyyy-MM-dd") +
"'",
  " 1 = 1 "
)
+
" AND MyCategory IN(" + "'" + dlg#TypicalExample#MyCategory.Replace(",
","', '") + "'" + ")"
```

When the TypicalWhereClause expression is tested, it brings up the dialog, and for the given values (seen above in dialog) it returns this:

1 = 1 AND MyCategory IN('Cars', 'Tools')

But if the "Use Start Date Filter" had been checked, it would return:

MyDate >= '2011-02-10' AND MyCategory IN('Cars', 'Tools')

The SQL statement would then be:

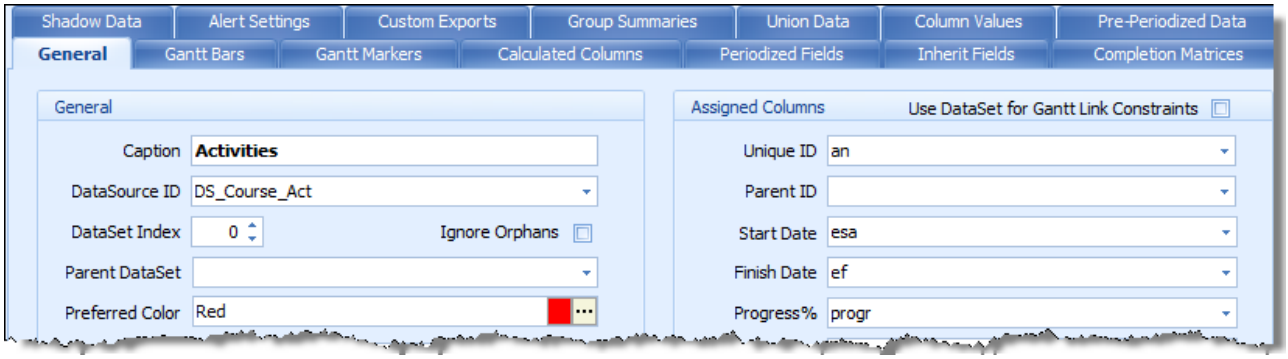
```
Select * from MyTable Where {TypicalWhereClause}
```

A separate chapter is dedicated to the design of Custom Dialogs.

Chapter 7: Setting up DataSet(s) and their internal relationships

A DataSet is the entity in Proteus that has the most diverse set of properties associated with it: Which DataSource to get data from, what additional columns to add to it, how data is visually presented, time-phased, exported, monitored, etc.

A DataSet in Proteus represents the data loaded from one SQL, combined with some internally generated fields, and some (optional) custom calculated fields. It also contains a host of analytical options, as well as presentation options.



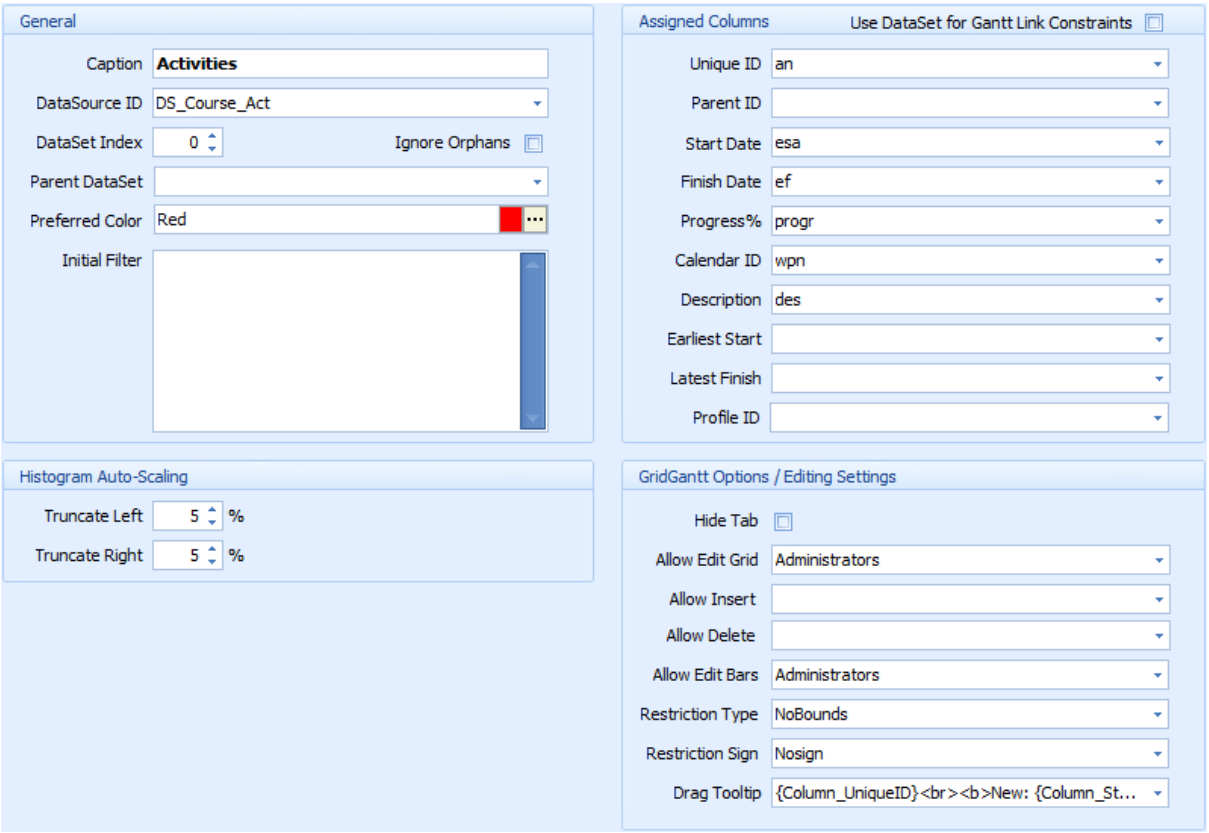
The various tabs where different types of properties can be set for a DataSet.

The topmost row of tabs indicates the various types of settings that can be made for a DataSet. Each of these will be given its own subchapter:

1. General
2. Gantt Bars
3. Gantt Markers
4. Calculated Columns
5. Periodized Fields
6. Inherited Fields
7. Completion Matrices
8. Shadow Data
9. Alert Settings
10. Custom Exports
11. Group Summaries
12. Union Data
13. Column Values
14. Pre-Periodized Data

Note: A number of input controls for a DataSet's values may be supplied as *"dynamic variables"* instead of supplying a fixed value. You may for example use a *calculated column* to control the color of a Gantt Bar. Instead of typing "**`blue`**" in the input control, you may instead type for example "**`{BarColor}`**", where **`BarColor`** is a calculated field that may return a value based on a few if-statements.

General



The tab where the General settings are defined for a DataSet

Typical Steps:

Step 1: Give the DataSet a Caption. This will be shown in Proteus' User Interface at runtime.
Step 2: Define which of the pre-defined DataSources to use.
Step 3: Define Assigned Columns (fields to use for particular functions):

Assigned Column	Description
Unique ID	The field that contains unique values (no duplicates). For hierarchical relationships, this is used as the parent field for all DataSets referring to this DataSet as the **Parent DataSet** (dropdown in top left area). The **Parent ID** (see below) field must then contain values that match one of the parent DataSet's **Unique ID** values.
---------- *entries below this point are optional* ----------	
Parent ID	(Optional) Explained in row above.
Start/Finish Date	(Optional) The fields used to control the Gantt Bar's start/end position, as well as start/end date for time-phasing.
Progress %	(Optional) The field used to control the Gantt's progress bar.
Calendar ID	(Optional) The field used to select which calendar to use.
Description	(Optional) The field used as descriptive text for an object. Special use: If a value in the Parent ID field does not match any value in the parent DataSet's Unique ID, it is considered an "orphan object" – in such cases the Description field value will be prefixed with the text "Missing Parent: <Parent ID>" (where <Parent ID> is replaced with the specific value).
Earliest Start / Latest Finish	(Optional) The fields used to control the "accepted timerange" of the object. These fields are used in conjunction with **Restriction Type** and **Restriction Sign** (at bottom right of form) to determine how "out of range" is handled and visually presented.

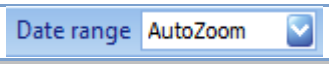
Profile ID	The field to use to select which Profile to use (from the loaded profiles)

***Note**: If you have completed all the "Typical Steps" described so far in this document, you may save it and run it from Proteus (or pressing "Launch"-button in Configurator's ribbon bar) – this represents the "Hello World" workspace in Proteus (i.e the minimum effort to get some results from Proteus).*

Advanced Settings:

Here are some explanations for the other controls on the form:

Control in Form	Description
General	
DataSet Index	This is automatically incremented, but can be edited. It is used when giving names to relations between DataSets. Example: If DataSet with index 0 has 2 child DataSets, having index 1 and 2, then the relation between DataSet 0 and 1 is named "**REL0_1**" and the other between 0 and 2 is named "**REL0_2**". It is recommended to build hierarchies from the top and down, so that higher index values represent lower level sets of data. The names of these relations must be specified in aggregated calculated columns when there are *more than one* child DataSet, e.g: Sum(Child(**REL0_1**).MyDS1Field) and Avg(Child(**REL0_2**).MyDS2Field) If you have only ONE child DataSet, this would also work: Sum(Child.MyDSField)
Ignore Orphans	If a DataSet, **C**, is a child of another DataSet, **P**, then a row will automatically be added to DataSet P with UniqueID "**Orphan_C**". This row will *not* be added if **Ignore Orphans** is checked for DataSet **C**. Important Note: The above is true is the UniqueID field is of "string"-type. For other data types the following will occur: **GUID**: A new GUID key will be created to act as the "Orphans' Parent". Its value will be **00000000-0000-0000-0000-000000000000**. **Numeric**: *Not supported*. You should check the "**Ignore Orphans**" in order to avoid error messages in log (built during load of Workspace).
Parent DataSet	(Optional) Used for hierarchical linking of DataSets, and indicates which DataSet to use as parent for the current one. Note that in this case you must specify which column to use to point to the values in the parent DataSet – in the "Parent ID" setting under the "Assigned Columns" section.
Preferred Color	(Optional) The color of Gantt bars when exporting to Safran Planner (see Appendix for more info). It is also shown in histogram's panel for setting visibility (hide/show) of its periodized fields.
Initial Filter	(Optional) The filter to use for this DataSet when Workspace is opened. This could refer to the result of an Expression, using the {MyExpression} syntax. Note that a filter for one DataSet will have a cascade effect (reducing the number of visible rows) for related child DataSet(s).
Histogram Auto-Scaling	
Truncate Left/Right	The percentages of scope (typically hours) on left and right "edges" to be truncated when automatically setting time range for histogram. This timerange will be used when a histogram has this setting in Proteus:

	
GridGantt Options / Editing Settings	
Hide Tab	Controls if the tab for this DataSet is visible or not. Typically this is the case when you are using a DataSet as visual link information (arrows showing relations between objects) and do not want to see the actual raw data in a grid.
Allow Edit Grid	Controls which Groups can Edit content of Gantt's grid (for those columns that are marked as "Updateable").
Allow Insert	Controls which Groups can add new rows when a grid for this DataSet is used in a DataLayout.
Allow Delete	Same as above, but controls the deletion of rows.
Allow Edit Bars	Controls which Groups can manipulate the Gantt's bars, using drag and drop.
Restriction Type	The type of "date-bounds" that an item in Gantt Chart should have for this DataSet relative its parent item's start/finish dates. The choices are: **LimitToStartBounds**, **LimitToFinishBounds**, **LimitToStartAndFinishBounds**, and **NoBounds**.
Restriction Sign	Controls if a "Stop Sign" icon is show when a restriction has been violated. Choices are: **Nosign** and **Stopsign**

Drag Tooltip Format String	The formatting to use when a Gantt bar is dragged in the Gantt chart. This is given a default value that in most cases works fine:
	`<size=14>Job:`
	`<color=blue>{Column_UniqueID}</color></size>`
	`Description: {Column_Description}`
	`<b>New Range: {Column_Start:d} - {Column_Finish:d}`
	`(Moved {Column_MovedDays} days) </b>`
	`Original: {Column_Original_Start:d} -`
	`{Column_Original_Finish:d}`
	
	As seen from format above, it is possible to refer to old and new values with the following field references:
	{Column_MovedDays} – The number of days moved.
	{Column_Start} – The new start date.
	{Column_Finish} – The new finish date.
	{Column_Original_Start} – the original start (at load).
	{Column_Original_Finish} – the original finish date.

References to assigned columns for a DataSet:

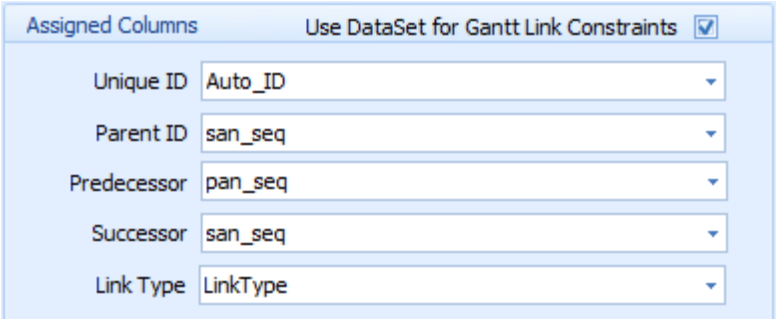
As shown in DataSet's General Properties, it is possible to assign which column to use as the UniqueID, Progress, Start, Finish and Description, etc.

In order to allow better default settings for some input controls (such as Gantt Tooltips, etc), it is convenient to refer to these fields using a generic reference, instead of a specific fieldname:

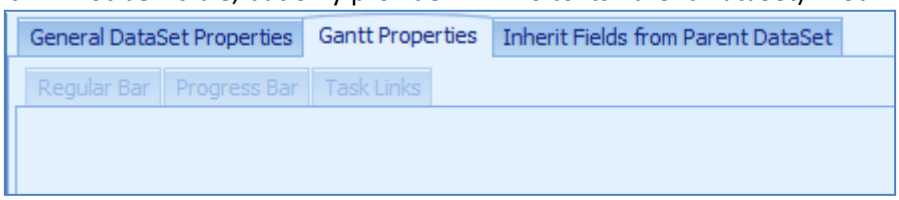
{Column_UniqueID}
{Column_Progress}
{Column_Start}
{Column_Finish}
{Column_Description}

Using a DataSet as Task Link definitions for another (parent) DataSet:

There is a checkbox at the rightmost corner of the form, and if checked, it will signify that the DataSet will be used exclusively to render Task Link information (in Gantt Chart) for the Parent DataSet.



When this is checked, you will not be able to control the visual Gantt Properties for this DataSet (since it will not be visible, but only provide link info to its Parent DataSet). You will then see this:

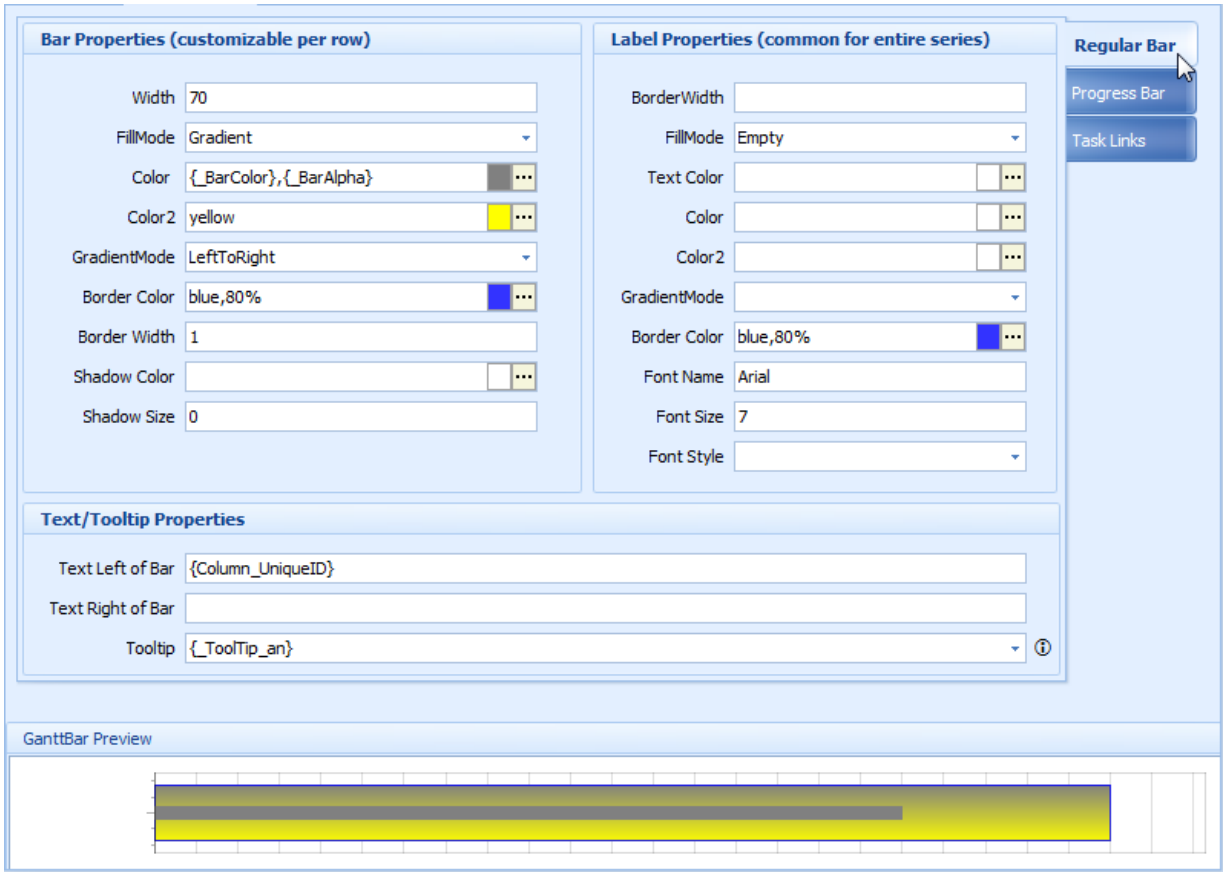


The function of each assigned column:

Assigned Column	Description
Unique ID	A field that contains unique values for each row of loaded data.
Parent ID	This field must contain values that match one of the parent DataSet's **Unique ID** values.
Predecessor	The Field that contains information on predecessor object (must match values in parent DataSet's Unique ID).
Successor	The Field that contains information on successor object (must match values in parent DataSet's Unique ID).
Link Type	Currently these link types are supported. The field must contain the (case sensitive) text of one of the following values: • **FinishToStart** • **StartToFinish** • **StartToStart** • **FinishToFinish**

The various planning systems have their own internal representation of link types. In Safran, for example, the "constraints" table's field "ntypec" has these settings:
1 = StartToStart
2 = StartToFinish
3 = FinishToStart
4 = FinishToFinish

Gantt Bars

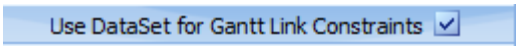


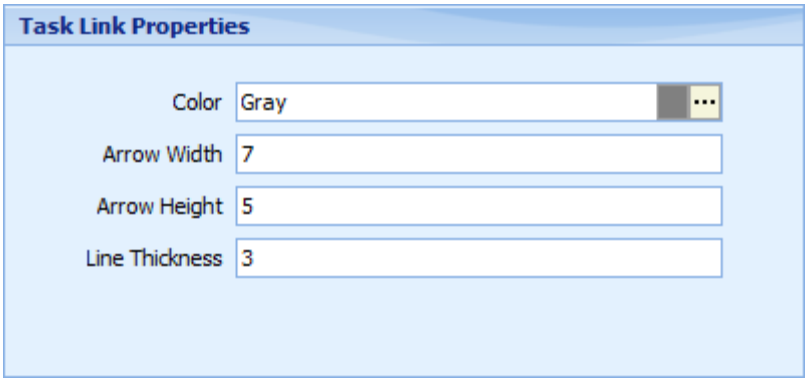
The form where Gantt Bar properties can be controlled (Regular Bar, Progress Bar, and Task Links)

As mentioned earlier in this document, and as seen in example above, it is possible to supply values wrapped in curly brackets – signifying a field value instead of a fixed value. This technique offers added flexibility when implementing functionality to a Proteus Workspace: Attributes in your data may easily be converted into something "sensory". In the specific example above, 2 separate calculated columns ("_BarColor" and "_BarAlpha") are used to control color, and opacity respectively. See subchapter on calculated columns further down in this chapter, as well as Appendix C: Syntaxes for assigning colors.

There are 2 other tabs for setting Gantt Bar properties, and the middle tab is virtually identical to the one shown above, only that it controls the progress bar properties instead.

The lowest tab controls the properties of the Task Links. They are only relevant to control, if the DataSet has a child DataSet that is marked as "Use DataSet for Gantt Link Constraints":

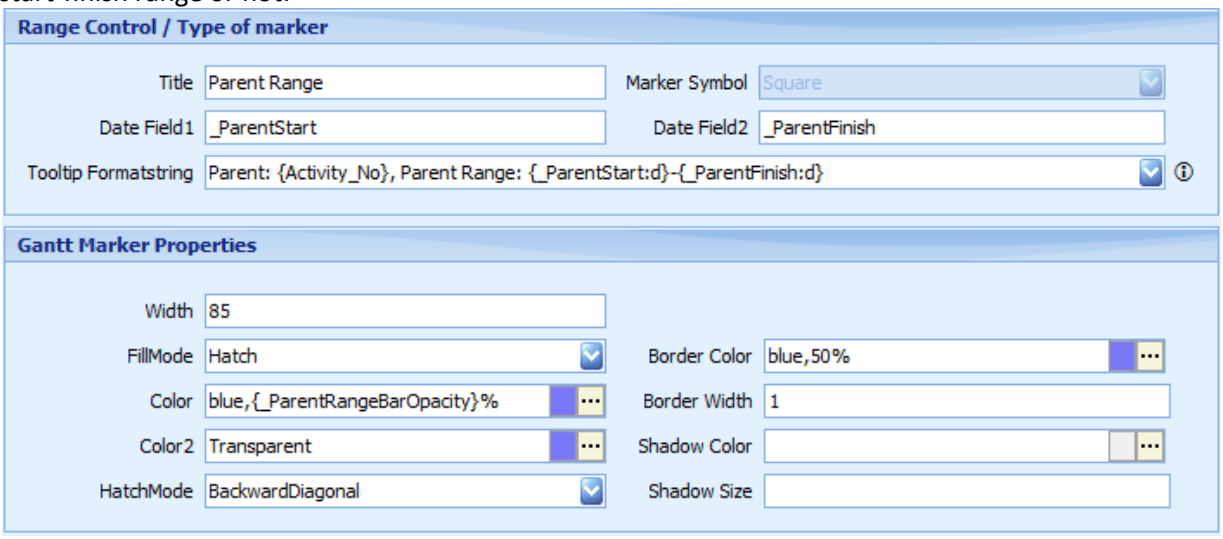




Gantt Markers

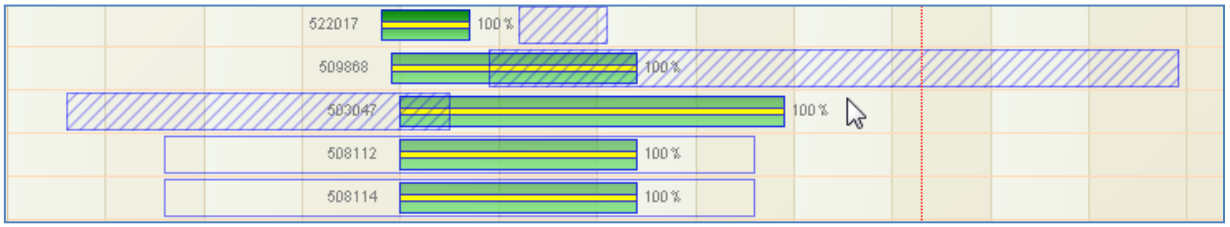
Each DataSet can have any number of Gantt Markers shown superimposed in its Gantt Chart.
A Gantt Marker can either be a symbol (star, square, etc) controlled by one date field (Date Field1), or it can be a bar controlled by 2 date fields: “Date Field1” for Start and “Date Field2” for Finish.

Example:
Below is an example of a bar that is intended to reflect the start and finish dates of its parent DataSet. It uses a Hatch brush, and we here also want to visually indicate if the object is within its parent’s start-finish range or not:



We see above that the color’s transparency (or rather opacity – the opposite of transparency) is controlled by a calculated column expression, {_ParentRangeBarOpacity}. Its appearance will therefore be controlled by the result of the calculated column, having this expression:

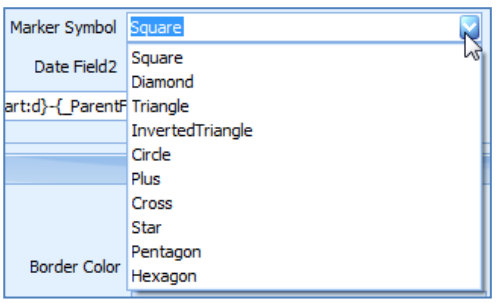
```
Iif( Start <_ParentStart OR Finish >_ParentFinish, 40, 0)
```



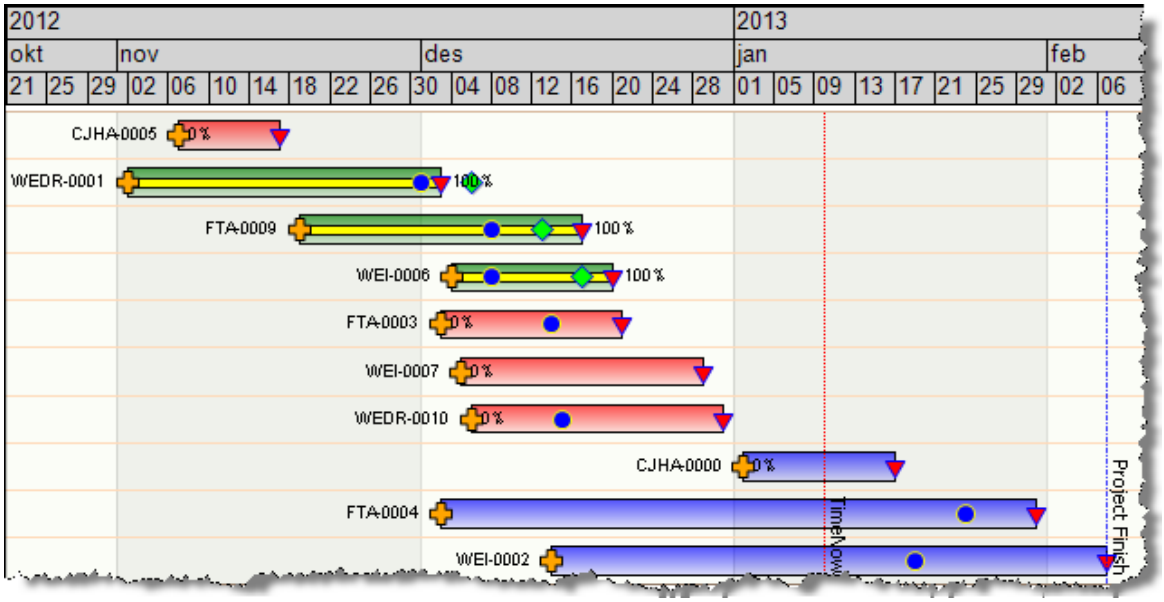
We see that the last 2 bars show no hatch brush, due to opacity 0 (full transparency) - and we also see that they are, in fact, within the parent’s start-finish range, whereas the others are outside and hence show the hatch brush.

The use of such calculated columns open up a wide range of possibilities to show attributes in data in a visual equivalents. In this way perceptional psychology can be made an integral part of the way things are presented to project personnel.

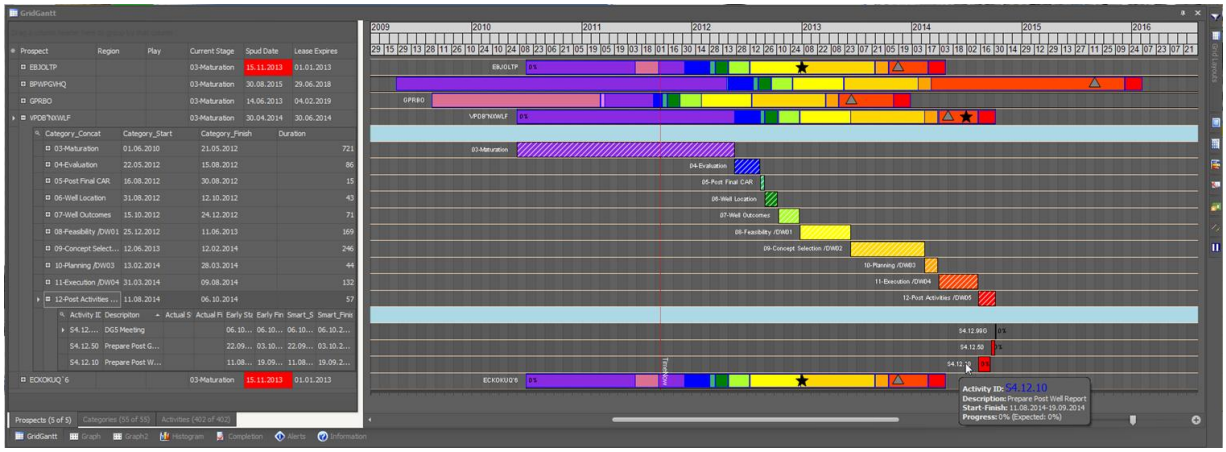
A note on the Width property:
The list of possible markers is shown at right, and in this mode, the Width property controls the size of the symbol, where the value 12 is a typical height (pixels). In the case of a range, the Width property indicate % of row height. A typical value in this case is 85.



Some examples of using Gantt Markers:

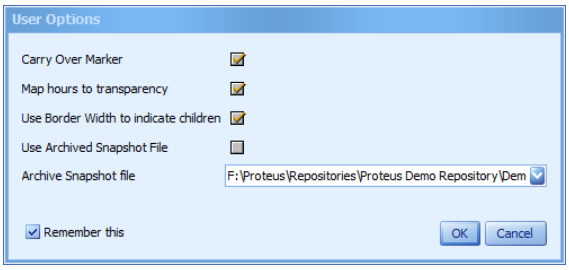


Important dates shown in bar (Created, Verified, Finished, Due Date)

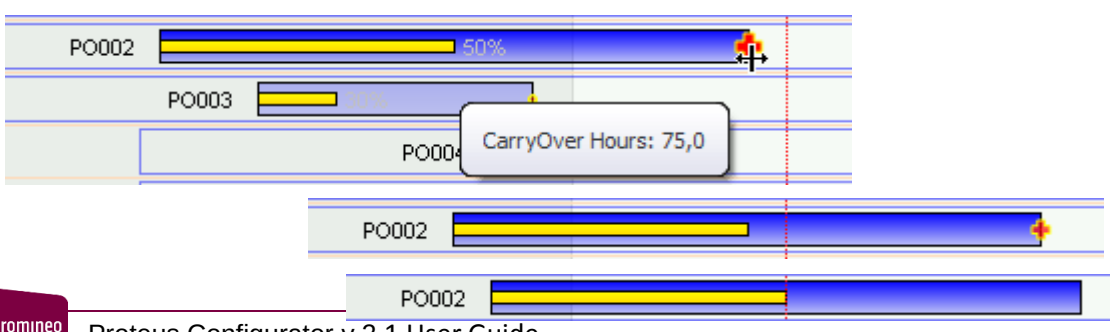


Each multi colored bar in reflects an entire network in planning system (Safran, P6 or other), and bars can be expanded to lower hierarchical levels (ending up with task level).

It is possible to use a custom dialog to allow user to control which Gantt Markers to show, and how to show them:

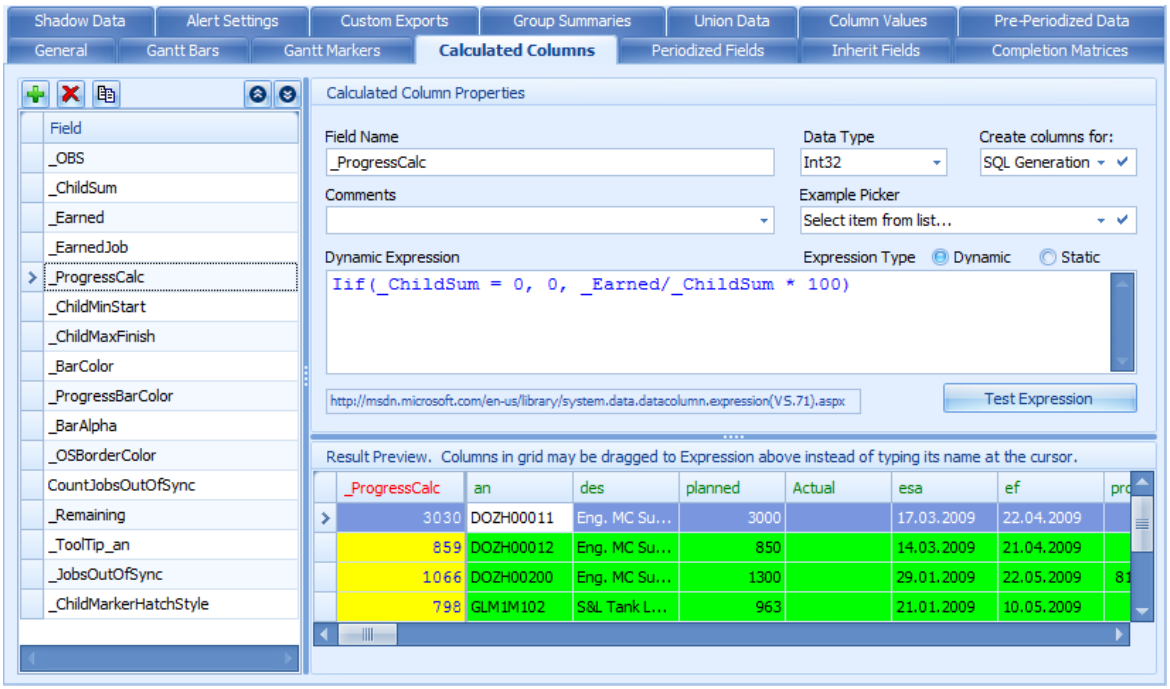


"Carry over hours" (work that should have been completed, but is not), can be calculated in Proteus, and visualized using a red cross who's size indicate the magnitude of carry over scope. When the bar is moved in time (drag & drop), you can see below how the carry over scope is reduced, and consequently how the Gantt Marker becomes smaller, until it vanishes when there is no carry over scope:



Calculated Columns

As seen in the example with Gantt Markers (previous pages), calculated columns can play a vital role in allowing Proteus to present things in a dynamic way, so that visual attributes can reflect specific properties in the loaded data.



Each calculated column will be "added" to the list of fields loaded from the DataSource's SQL. It is possible at runtime (in Proteus) to filter out the fields that are calculated from those that are given from SQL, as well as some other categories of fields.
The calculated column must be given a **Field Name**, and the **Data Type** of the result must be specified:

DataType	Description
String	A text values of any length
DateTime	A Date (and time) value
Int32	An integer
Decimal	A decimal number (lower precision than Double)
Double	A decimal number with high precision (to be preferred over Decimal type above)
Boolean	A true or false value (binary). Will show as a checkbox if used in a Grid colum

In the top right area of the above screenshot, one can see a choice for "**Expression Type**". These two types need to be described separately:

Dynamic Calculated Columns

Uses the technology provided by Microsoft to add computed columns to a so called DataTable object. The supported types of expressions are described in this document from Microsoft:
http://msdn.microsoft.com/en-us/library/system.data.datacolumn.expression(VS.71).aspx

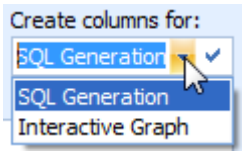
As the name implies, the dynamic column always recalculates its value when data changes in a row. It has a very high performance, so quite a number of calculated columns can be defined without any noticeable performance hit.
A calculated column can refer to another calculated column (as seen in example above).
A feature that is extremely useful in the dynamic column is its ability to refer to child and parent data through an established hierarchical relationship (see earlier chapter how they are defined in Proteus).

Some Examples:

- `Parent.FieldX + 'abc'`
- `Sum(Child.NumericField1) + 1,` or if more than one child-relation: `Sum(Child(RelationID).NumericField1) + 1`
  In Proteus, a relation between DataSet with index 0 and index 1 is named "REL0_1":
  `Sum(Child(REL0_1).NumericField1`
- `Iif(MyField > 10, 'Big', 'Small')`

Please note that in calculated columns you need to wrap strings in single quotes, whereas in Expressions you would need to use *double quotes*.

Note that there is a dropdown with an "insert button" (the check symbol to right of dropdown arrow) that allows inserting standard settings for different types of situations:



Calculated columns inserted for "SQL Generation":

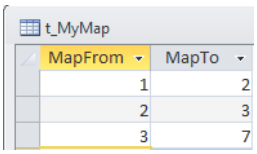
Column Name	Description
_ UpdateSQL	'UPDATE [BASETABLE] SET [FIELDNAME] = [FIELDVALUE] [UNIQUEWHERE]'
_ InsertSQL	'INSERT INTO [BASETABLE]([FIELDNAMES]) VALUES([FIELDVALUES])'
_ DeleteSQL	'DELETE FROM [BASETABLE] [UNIQUEWHERE]'

The replacement markers, enclosed in square brackets above, are replaced by Proteus at runtime with appropriate values. If you, for whatever reason, want to customize these statements, it is quite possible to do so manually. More info on this can be found in Appendix B.
Also see section "Fieldnames with special meaning" on next page.

Calculated columns inserted for "Interactive Graph":
These all start with "**_CustomDraw**", and are described in more detail in the chapter called Controlling Chart details using special calculated column names.

The {{Global Expression}} syntax

It is possible to embed a global expression in a calculated column, and use *its evaluated result* (always a string) as the actual formula for the calculated column.
This can be especially useful if you have the following scenario:
You want to test if a value in a field, **FieldX**, matches values found in a table, and if so, a mapped value in that same table should be returned (according to the table "t_MyMap" having 2 columns:


t_MyMap

MapFrom	MapTo
1	2
2	3
3	7

If this mapping was fixed, you could solve the challenge using a nested Iif-statement like this:
`Iif(FieldX=1, 2, Iif(FieldX=2, 3, Iif(FieldX=3, 7, FieldX)))`

But in our case, this was not an option, since we wanted to control the mapping from a database table, "MyMap". Here are the steps to achieve our goal:

1. Create a DataSource called "**DS_MapTable**" with SQL: "**Select * From t_MyMap**"
2. Create an Expression called "**MyDynamicIif**" and give it the following content:
   `General.GetDynamicIifExpressionFromDataTable(ds#DS_MapTable.GetDataTable(true), "FieldX", "FieldX", "MapFrom", "MapTo")`
   The result (when pressing Test button) would in fact be identical to our manual one above:
   IIF([FieldX] = 1, 2, IIF([FieldX] = 2, 3, IIF([FieldX] = 3, 7, FieldX)))

3. Set the formula in the calculated column to: **{{MyDynamicIif}}**

Static Calculated Columns

Although extremely useful and well-functioning, one curious "weak spot" with the (Microsoft) technology behind dynamic column expressions is that it cannot calculate the difference (in days) between two date fields, nor add N days to a given date field value… This was one of the reasons for expanding the possibilities into what we call "Static" calculated columns – that are only calculated *once* at load time, and hence the name "Static".
They generally take longer time to evaluate than dynamic ones, so should be used only when necessary – such as in the case with date column calculations. See next section for more info and examples.

The static calculated columns give access to a subset of functionality found in the .NET C# language, as well as some functionality created specifically for Proteus.
The documentation for this is virtually identical to the documentation for defining Expressions (that can be reused between Workspaces). The chapter "Defining Expression" gives an insight into these possibilities.

Fieldnames with special meaning

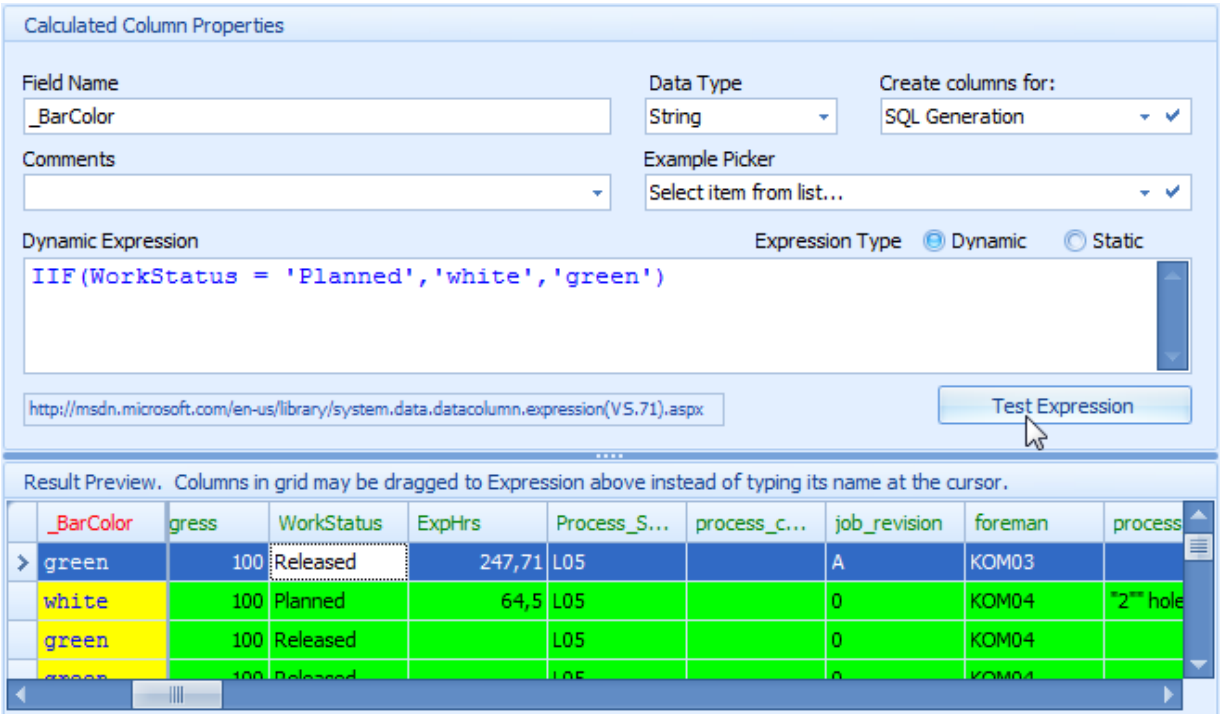
There are some fieldnames in a DataSet that carry a special meaning, and the table below gives an overview of what they are. They can be created in SQL statement or in calculated columns, depending on what is most convenient (the latter of the two is generally recommended):

Fieldname	Description
ToolTip<Fieldname>	Controls the format of a ToolTip for a given field in a Grid. It must be a string, and the fields you want to have replaced at runtime (when hovering over column) must be wrapped in curly brackets (e.g: "The tooltip for MyField is: {MyField}."). In this way tooltips can be assigned to individual columns in a Grid. Note that you have some markup options if you wish to have parts of the text colored or with different font sizes, bold text, etc.
CellFormat<Fieldname>	This allows setting a customized runtime cell-format for a cell in the column "<Fieldname>". The syntax for setting the formatting is described in Appendix. **Note**: With the introduction of Advanced Conditional Formatting (in Proteus v1.6), you may consider using it as an alternative to this technique.
ColumnHidden<Fieldname>	Allows hiding a column in grid.
ColumnCaption<Fieldname>	Allows forcing a caption on a grid column header.
ColumnToolTip<Fieldname>	Allows forcing a tooltip text on a grid column header.
_UpdateSQL, **_InsertSQL**, **_DeleteSQL**	These can also be specified for a particular column by appending "_<Fieldname>" as seen above. Then they will be used to customize update/insert statement for individual field. Appendix B gives more info on these here.
_CustomDraw...	Click here for more info.

Testing a calculated column expression

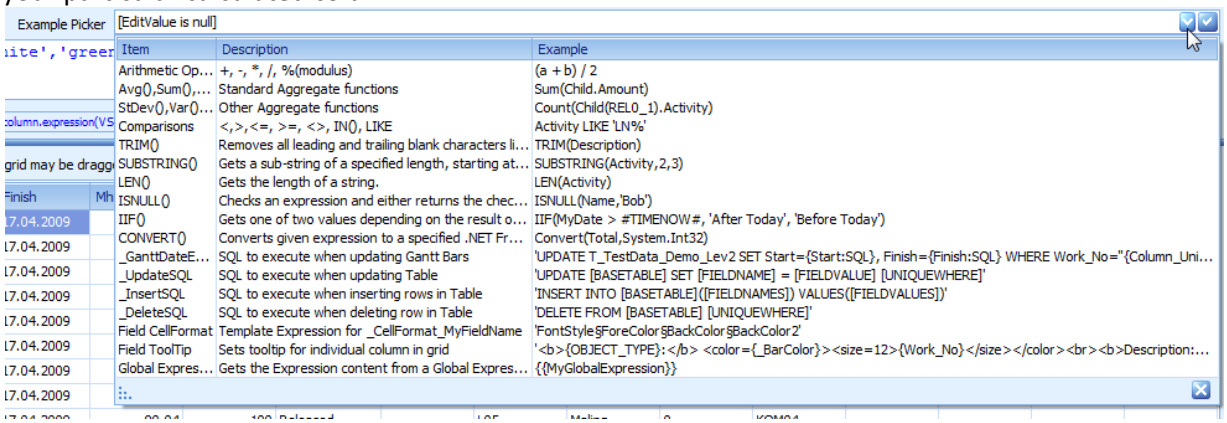
In the user interface, you can drag and drop fields from the grid (showing persisted data from the DataSource used in the DataSet) onto the "Expression" textbox (but note: the dragged text will always be inserted *at the* cursor and *not where it is dropped by mouse*). By pressing "Test", you will see a yellow column at left part of grid reflecting the calculated value. If an error is found, it will be stated as a red text under the grid.
Useful Note: You may ***edit*** the column value in the grid to provide varied input condition for your expression and the result will (for dynamic calculated columns) *instantaneously* change to reflect the result.

Note: This test function is not "fool-proof" in that you may not run hierarchical (like Sum(Child.MyField)) calculations, since the relationships will be built by Proteus at runtime. Even though it tries to simulate values for hierarchical formulas, be aware that the Test button may result in error messages here, while they may *not* occur in Proteus. The final test for any calculated column is done at runtime.



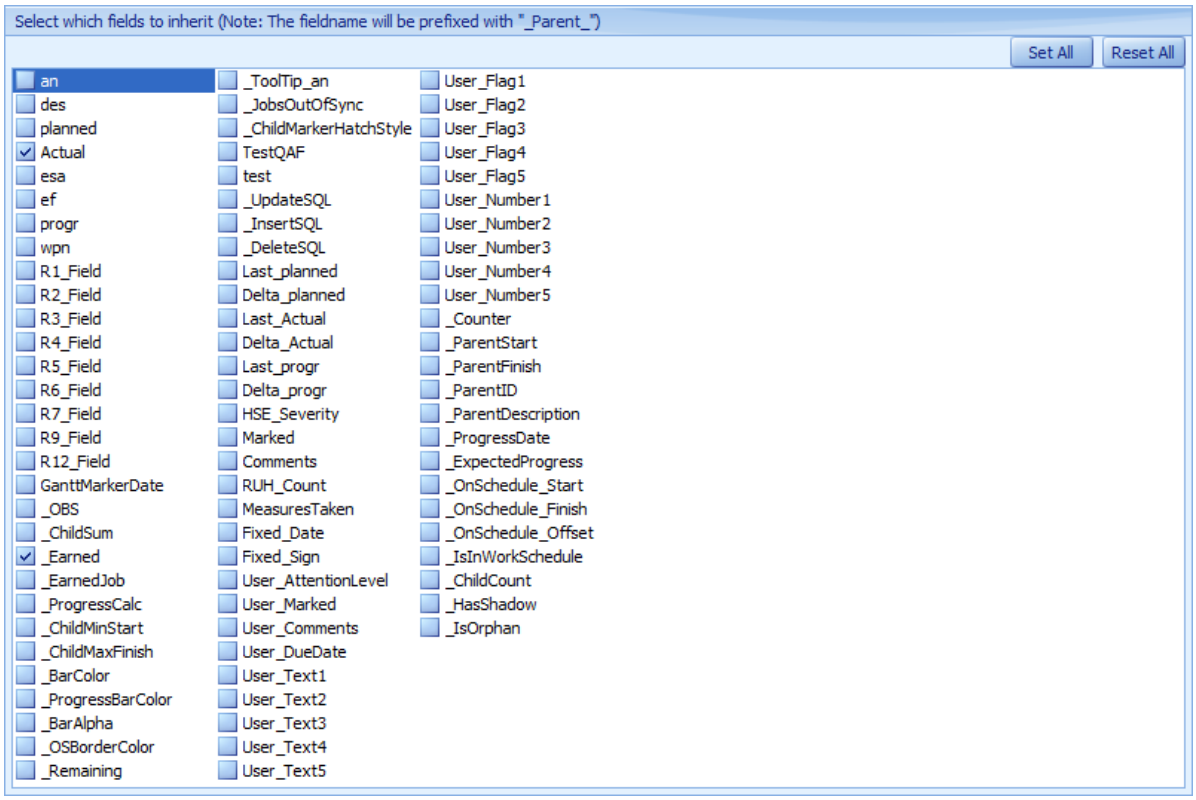
The resulting calculated column shown in yellow, "SQL columns" in green, and "Internal column" in gray.

There is an example picker in the Configurator, allowing examples to be inserted as starting point for your particular calculated column:



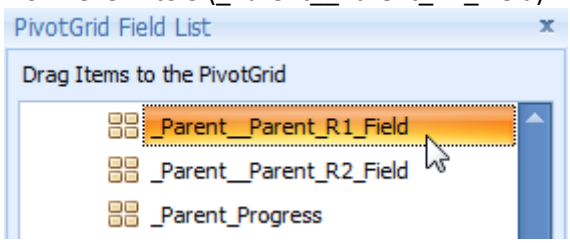
Inherited Fields

A DataSet that is a *child* of another DataSet will show a list of the parent's fields, and you can check which of these fields to "inherit" down to the child DataSet:



This can be particularly useful when doing Pivots (see section on Pivot Tables in *Proteus User Manual*) at the lowest level of hierarchical data, by allowing values from higher up in the hierarchy to "trickle down" to this level. The fieldname for such an inherited field will be prefixed with "**_Parent_**".

Note that it is possible to do multiple levels of inherit, and the field's name will then have as many "_Parent_" prefixes as inherited levels. Example from Proteus Pivot where fields have been inherited from level 1 to 3 (_Parent__Parent_R1_Field):



Periodized Fields

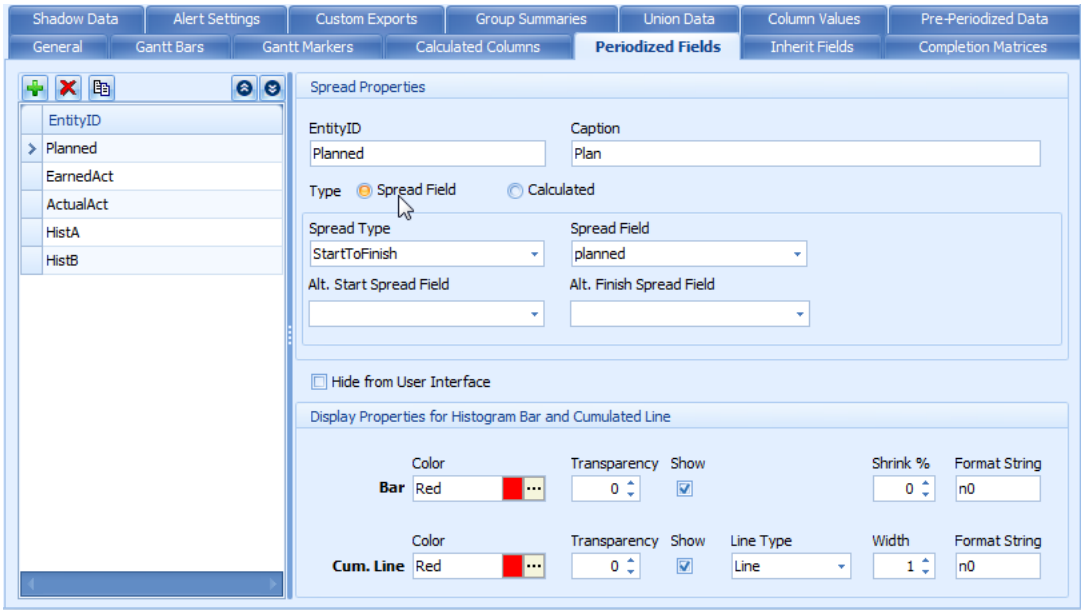
If you wish to time phase (also called “periodize” in this document) a particular field, i.e spread its numeric quantity over the dates that it is defined for (start and finish) – then the field has to be added to the list of Periodized Fields. There are these alternatives to the normal **StartToFinish** spread:

SpreadType	Description
StartToStart	The entire quantity is placed on Start date.
FinishToFinish	The entire quantity is placed on Finish date.
StartToTimeNow	Spread from Start date to TimeNow(*) date. Typical use is for “earned” hours – they cannot be earned ahead of today’s date.
TimeNowToFinish	Spread from TimeNow(*) date to Finish. Typical use is for “remaining” hours – they should by definition start from today and spread forward in time.
Periodic	The spread is made externally and Proteus’ periodization engine is bypassed. See next chapter on “Pre Periodized Data” for more info.

(*) TimeNow = the date defined at Workspace level to be “Today’s date”. More info here.

It is possible to specify which fields to use as Start and Finish for spreading, but if omitted, the Start and Finish fields assigned to the DataSet are used (link here).

It is possible to control several visual attributes for each periodized field: Which color it should have, what transparency it should have, if it should be shown initially, and what width-shrinkage percentage it should have (so that a bar can be shown inside another bar). Most of these options can be controlled for the cumulated line as well:



The form for defining properties for periodization (time phasing).

If you want to hide a row (periodic and cumulated) from the user, it is possible to check the "**Hide from User Interface**" box. This may be of interest if a timephased row is used as an intermediate step in a calculation to give the final result to user (see next page).

Note: As s general rule, no date that is marked as Non-Working can receive any spread result. There are however two exceptions to this rule:

1. If the calendar in question has ***zero working days*** in the assigned spread time-span, then the quantity is spread equally on all of these (non-working) days.
2. For Pre-Periodized sources

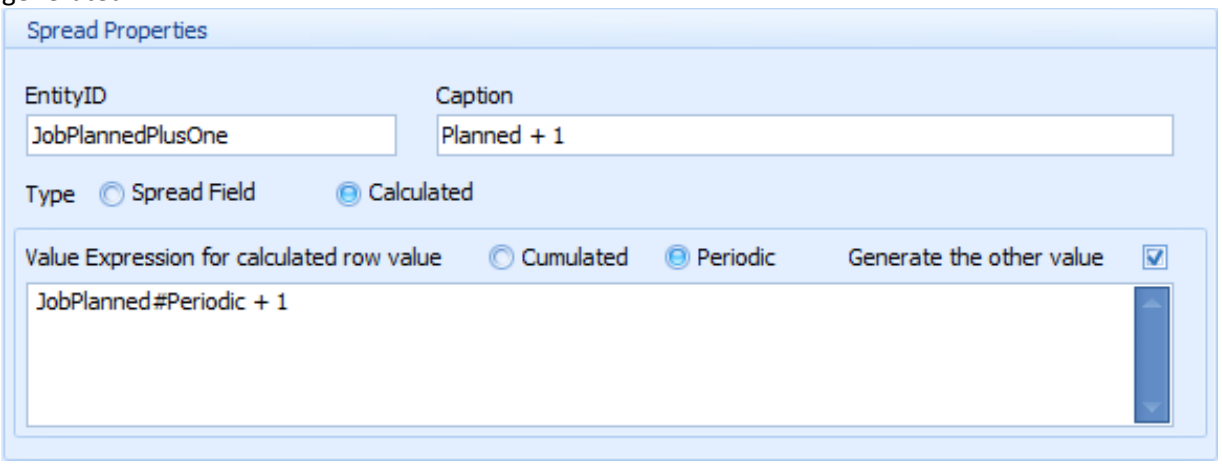
Generating rows in histogram data from formula

In some cases you may need to add calculated "rows" of data to the periodized result set. This is possible by choosing the "Calculated" option in the radio-button called "**Type**".
You may then write the formula to use for producing the new row (periodic and/or cumulated) of data, based on the value in other rows. The syntax for such a formula conforms to the rules for an Expression. You reference an existing row of data by typing the Periodized Field's **EntityID**, followed by any of these suffixes:

Suffix	Description
#Periodic	The periodic value.
#PeriodicPercent	The periodic value relative the maximum periodic value in row (see below).
#PeriodicMax	The maximum periodic value.
#Cumulated	The cumulated value.
#CumulatedPercent	The cumulated value relative the maximum cumulated value in row (see below).
#CumulatedMax	The maximum cumulated value.

You may indicate if the formula should result in a **Periodic** or **Cumulated** row of data, and if "*the other*" (Cumulated or Periodic respectively) should be generated – by checking the box called "**Generate the other**".

Here is a very simple example of a calculated row, where both the Periodic and Cumulated rows are generated:

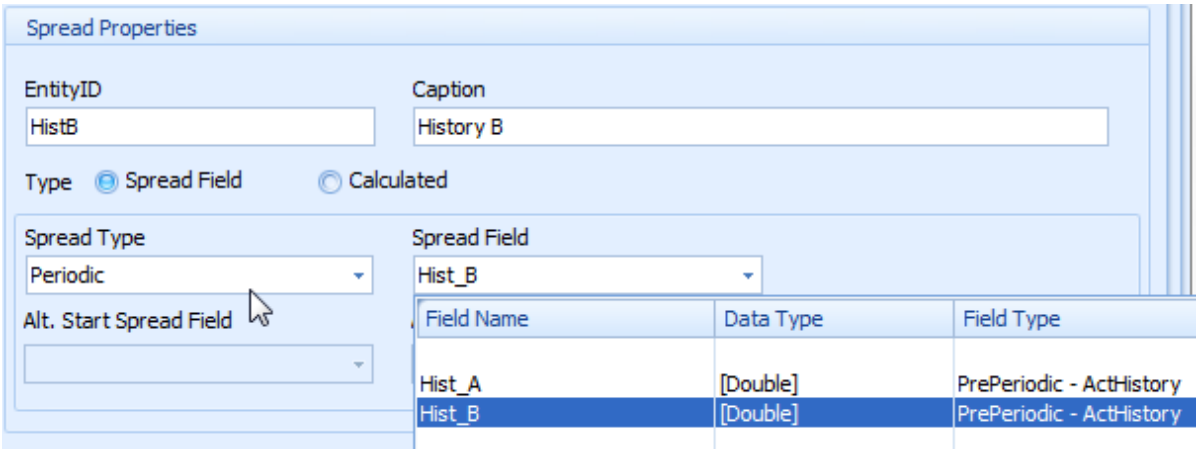


A simple example of a calculated row in histogram showing value of "Planned plus one".

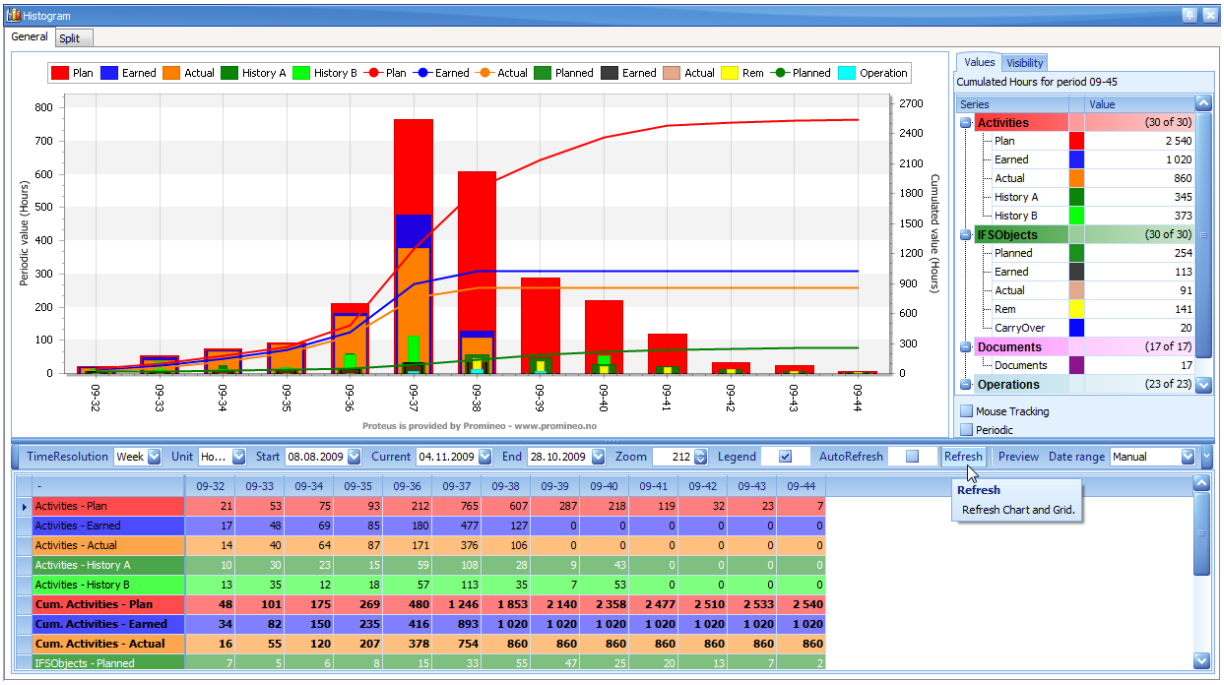
Important Note:
Since the periodized fields from ALL DataSets end up in the histogram's "data table" (the grid under the histogram), it is important that ***all EntityID:s are unique across all DataSets***, in order to ensure unambiguous references in formula expressions. A formula may reference rows from different DataSets.

Bypassing the Periodization Engine with pre-periodized data

If the Spread Type is set to "**Periodic**", there will be a different dropdown to select from in **Spread Field**, namely the ones given in DataSources marked as "**PrePeriodic**" (see next section):

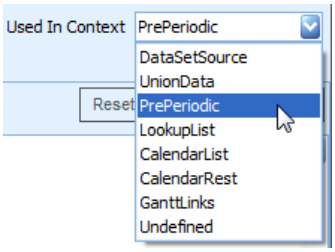


The field list In "Spread Field" will look different if "Periodic" is chosen as Spread Type.



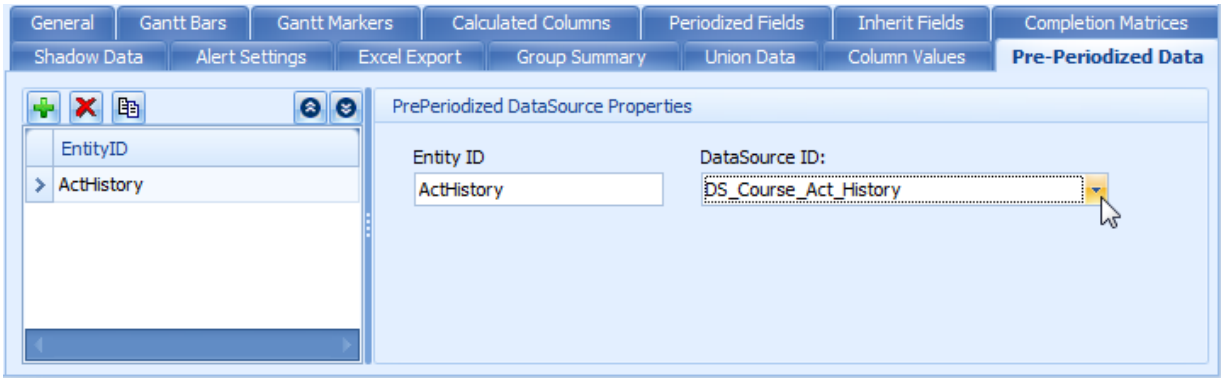
Example of how the above settings may look in Proteus at runtime.

Pre-Periodized Data



The last subchapter indicated that one type of "periodized fields" can be "*PrePeriodic*", i.e it is given by a DataSource (SQL) that returns pre-periodized data that bypasses Proteus' periodization engine, and shows up directly in histogram. Such DataSources have been marked as "**PrePeriodic**" (see dropdown for DataSource at right):

Any such DataSources can here be tied to an internal **Entity ID** ("ActHistory" in the example below), and then made available to the **Periodized Fields** form.



The format of a pre-periodized SQL must return these fields, where fieldnames may be arbitrary, as long as the internal order is maintained:

1. **ID-Field** (matching values found in DataSet's UniqueID field)
2. **Date-Field** containing the *date* to add periodic result to
3. **Value-Field1** containing the quantity to add to histogram at the above date
4. Value-Field2 (if any)
5. …. (Any number of Value-Fields here)
6. Value-FieldN (if any)

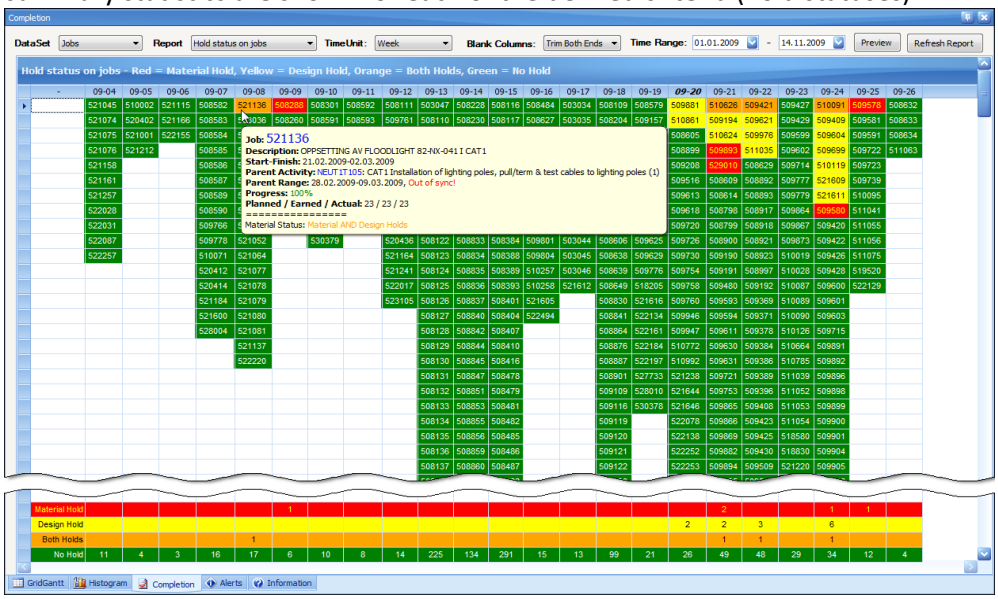
Completion Matrices

There is a reporting format that can be extremely useful in situations where one needs a “Completion Perspective” on a set of data.

An example from real life:

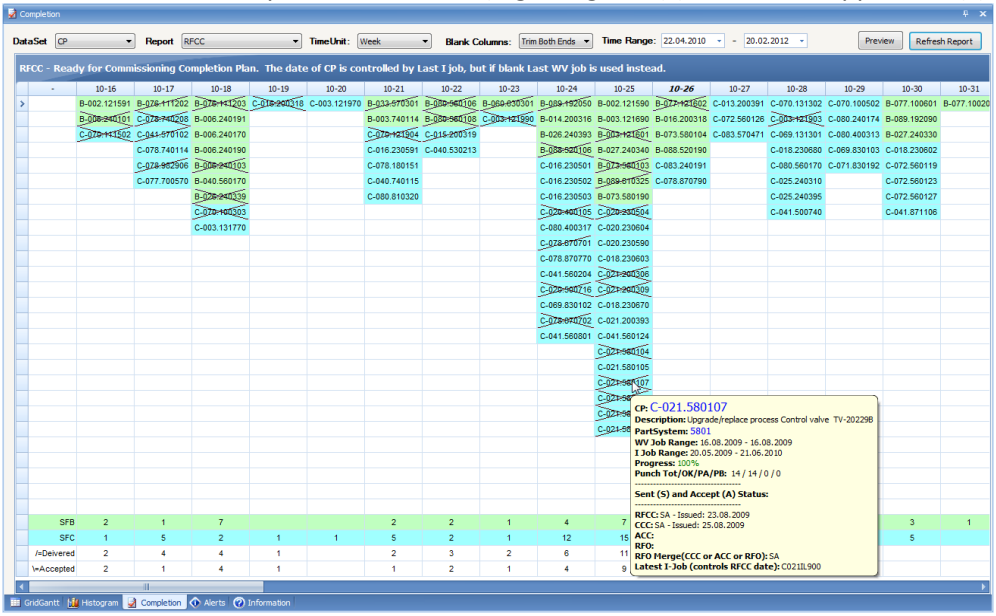
The Kashagan Project (one of the largest Oil & Gas exploration projects in the world, in Kazakhstan, on the Caspian Sea) needed an overview of all job cards that had material and design hold, i.e. the job could not be carried out because the required materials were missing or the drawings needed to carry out the job were not complete.

It is quite expensive for a welding crew to mobilize on a job, just to find that the pipes needed to do the job are missing. In the output below, each week shows a vertical set of jobs that all ***start*** in this week, and the coloring indicate the "hold status" for the job, where **red** means “*material hold*”, yellow means “*design hold*” and orange “*both*”. Green means “*no hold*”. At the bottom of the report, summary statistics are shown for each of the defined criteria (hold statuses):


Completion
DataSet Jobs
Report Hold status on jobs
TimeUnit: Week
Blank Columns: Trim Both Ends
Time Range: 01.01.2009 - 14.11.2009
Preview
Refresh Report
Hold status on jobs - Red = Material Hold, Yellow = Design Hold, Orange = Both Holds, Green = No Hold
Job: 521136
Description: OPPSETTING AV FLOODLIGHT 82-NX-0411 CAT1
Start-Finish: 21.02.2009-02.03.2009
Parent Activity: NEUT.IT.105: CAT1 Installation of lighting poles, pull/term & test cables to lighting poles (1)
Parent Range: 28.02.2009-09.03.2009, Out of sync!
Progress: 100%
Planned / Earned / Actual: 23 / 23 / 23
Material Status: Material AND Design Holds
Material Hold
Design Hold
Both Holds
No Hold
GridGantt
Histogram
Completion
Alerts
Information

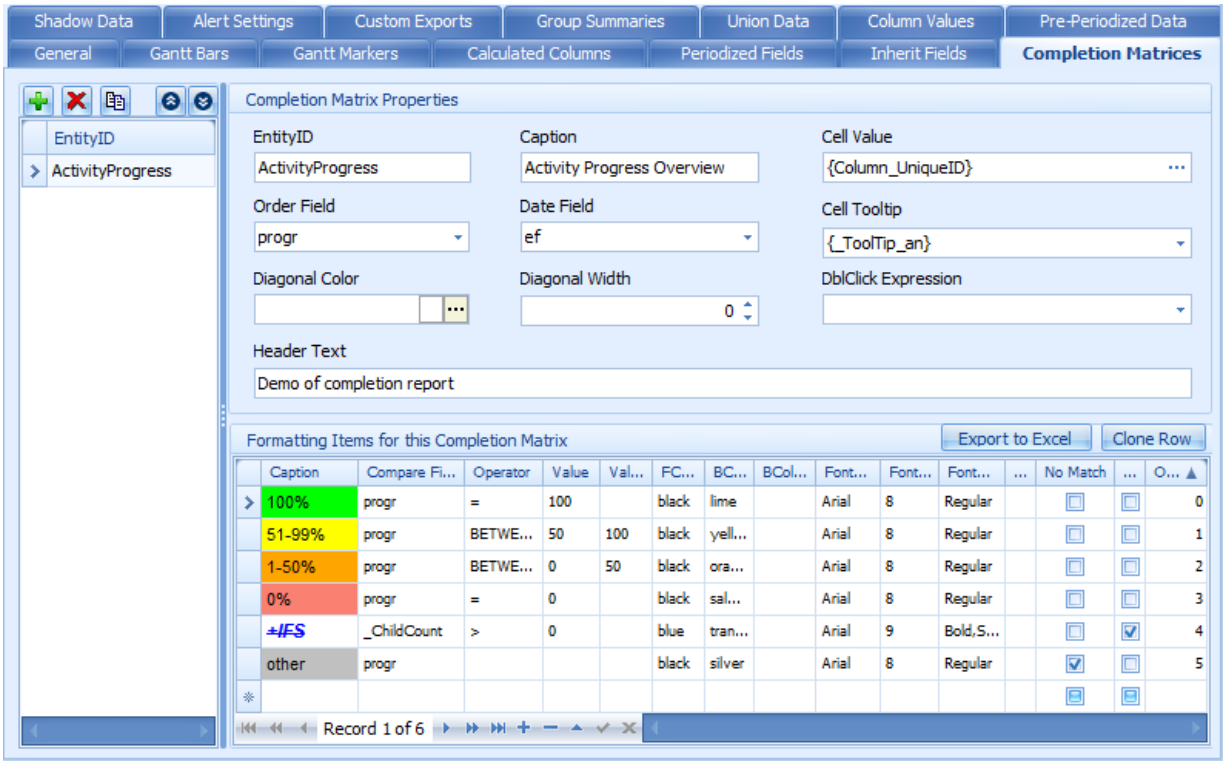
An example where red means “material hold”, yellow means “design hold” and orange “both”. Green means “no hold”.

Another very typical example of usage is for Mechanical Completion. Note how diagonal lines can be used to indicate an important condition (e.g: diagonal (/) = "Sent for approval", (\) = "Accepted")



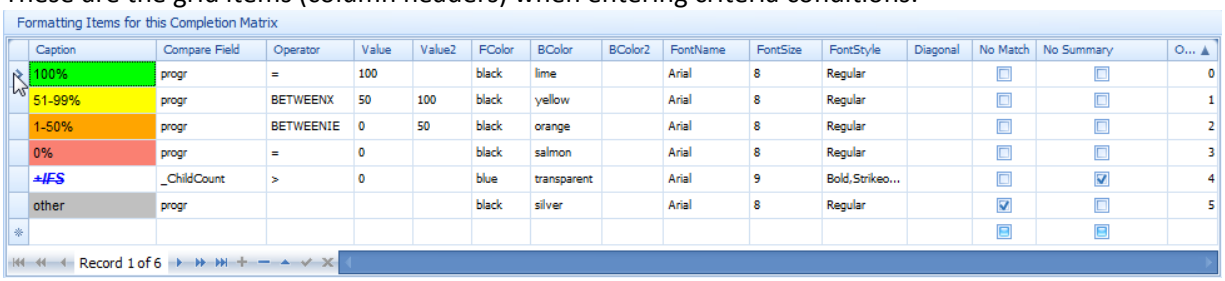
Another example where the Completion Matrix format is quite useful.

For a Completion Matrix (CM), we need to define some general settings (top area) and then we also need to define the (arbitrary number of) criteria we wish to highlight. Let's start by looking at the general settings:



Control	Description
EntityID	The unique ID for this Completion Matrix (CM)
Caption	The text to show in the dropdown for selecting which CM to use.
Cell Value Formatting	The format string controlling what to show in each cell. Field names may be wrapped in curly brackets to pull their value(s) at runtime.
Order Field	The field that controls the internal vertical order for each column.
Date Field	The field that controls which column the cell should be placed in.
Cell Tooltip Formatting	The format string controlling the tooltip. It is often useful to refer to a calculated field for this value, so that markup (colors and fonts) can reflect conditions in your data. E.g: {_ToolTip_MyField}
Diagonal Color	The color to use for diagonal lines.
Diagonal Width	The width of the diagonal lines.
DblClick Expression	The Expression to evaluate when cell is double clicked. This may result in opening a hyperlink to a related web page, sending a mail, etc...
Header Text	The text to show in the header section of the matrix at runtime.

These are the grid items (column headers) when entering criteria conditions:



And here are some words about the significance of each column in screenshot:

Control	Description
Caption	The text to show in the summary row
Compare Field	The field used for testing against given value(s) – see next 3 items below.
Operator	The operator to use for testing against value – see next 2 items below. The operators are presented in a dropdown, and some explanations are given in table below this one.
Value	The value to test against (see the 2 previous rows).
Value2	Used only when the Operator is one of the "Between" types.
FColor *	The foreground color to use.
BColor *	The background color to use.
BColor2 *	(Optional) Used if the background color should show a gradient transition.
FontName	The name of the font (family) to use
FontSize *	The font size
FontStyle	The font style (select from: regular, bold , italic, underline, strikeout)
Diagonal	The direction of the diagonal: / or \
NoMatch	Check if the criteria should be used if no other gets a "hit"
NoSummary	Check if this should be omitted from summary statistics.
Order	The sort order in grid - also controls the order of summary rows.

* Optional Syntax described in detail in section called "Creating a Continuous Coloring Scheme"

List of available operators:

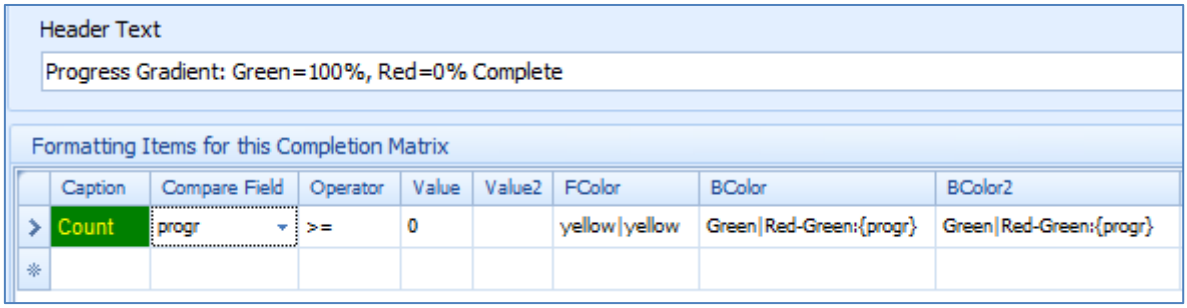
Control	Description
=, >, >=, <, <=, <>	Simple comparisons
IS BLANK, IS NOT BLANK	Test is blank or not
BETWEEN	See next row.
BETWEENIB	Include Beginning of interval - same as BETWEEN, included for completeness
BETWEENIE	Include End of interval.
BETWEENI	Include both endpoints of interval
BETWEENX	Exclude both endpoints of interval
BEGINS	Test if field-value begins with a particular string combination.
ENDS	Test if field-value ends with a particular string combination.
CONTAINS	Test if field-value contains a particular string combination.
IN	Test if field-value is one of the items in list.
LIKE	Test if field-value matches a "like" criteria with a particular string combination. The thing that is actually tested is a regular expression where a "^" has been added to beginning of the RegEx test.
PATTERNMATCH	A more sophisticated way to apply regular expression testing on the field-value. Further info on regular expressions can be found on the web – here is one good example: http://www.regular-expressions.info/examples.html

Most (from 3rd row and down) of the above operators are available with the "NOT" prefix to imply the opposite condition.

Creating a "Continuous Coloring Scheme"

There is an optional syntax for some of the fields in the Completion Matrix formatting items, which allows for defining *ONE* (and only one) row that results in a "Continuous Coloring Scheme".
This syntax can be set for the FColor, BColor, BColor2, and FontSize columns:
<LegendDisplaySetting>|<CellsEvaluatedDisplaySetting>

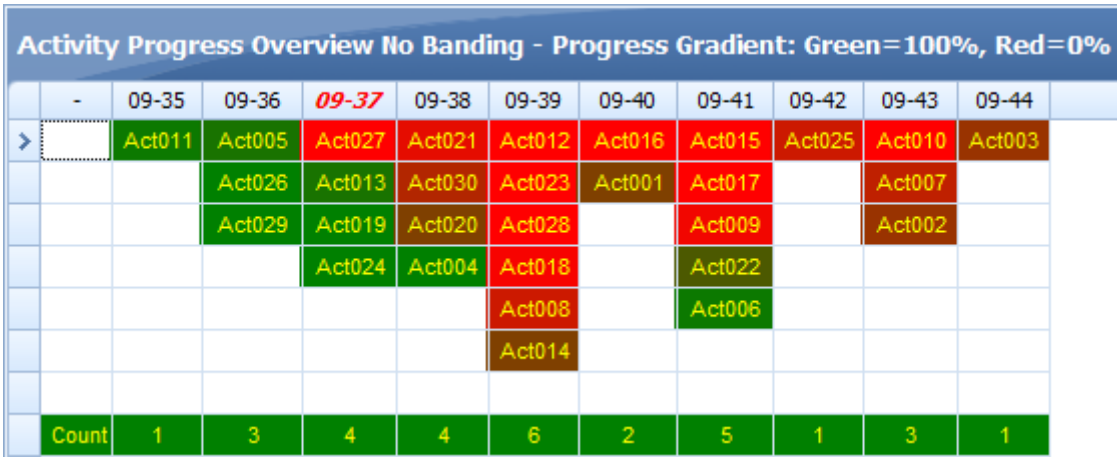
The text to the left of the "|" character is used for setting the legend properties, and the right part is evaluated using the known variable syntax, where fields in curly brackets get replaced with their actual value at runtime.
The example below may be useful for seeing how this is done in real life:



Example of a setting up a "Continuous Coloring Scheme" in Configurator.

More info on coloring syntax (the color interpolation syntax used above) is found here.

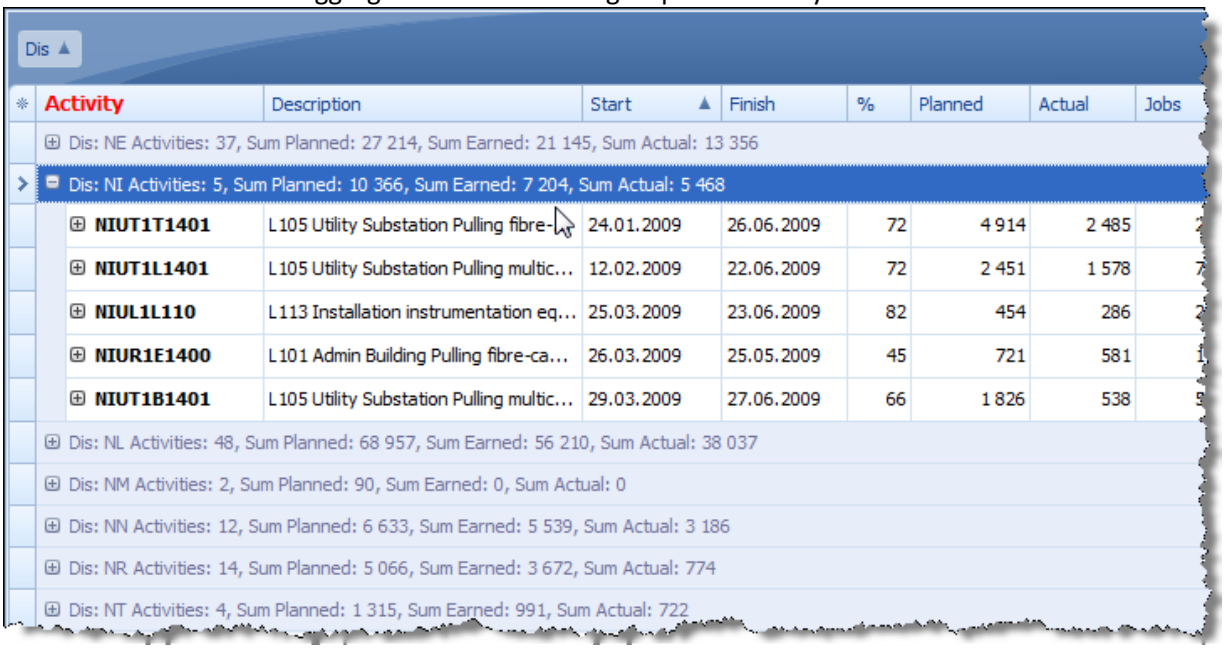
The result looks like this:



Example of the same "Continuous Coloring Scheme" seen when running Proteus.

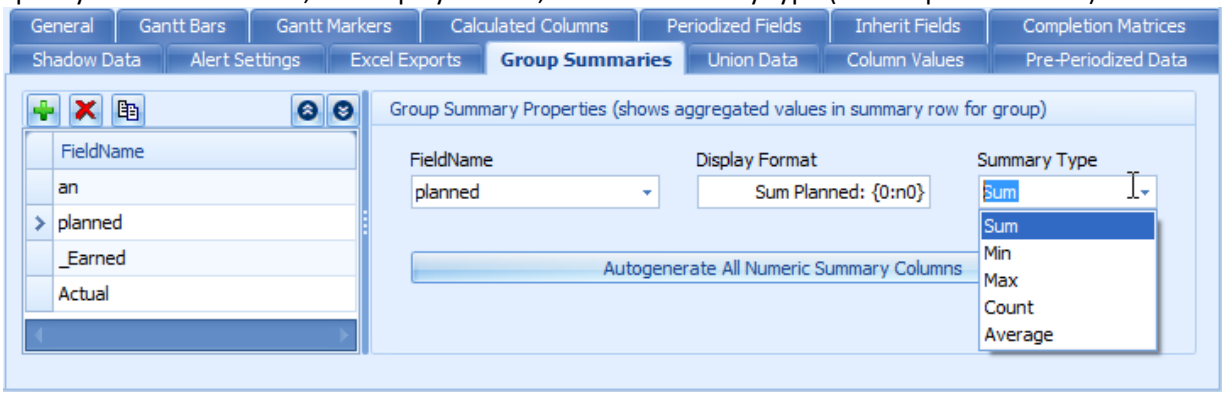
Group Summaries

It is often useful to show aggregated data for each grouped summary row:



The result in grid at runtime after setting up group summaries for a DataSet

You can add as many items as you like in such a Group Summary, and for each item you need to specify which field to use, the display-format, and the summary type (see dropdown below).



By pressing the button “Autogenerate All Numeric Summary Columns”, you will generate entries for all numeric fields. The Display Format will be “Sum <Fieldname> = {0:n0} and Summary Type will be “Sum”. Beware that this may result in a lot of columns that you may not want, and it might take more time deleting them than adding the few ones you needed in the first place.

Note:

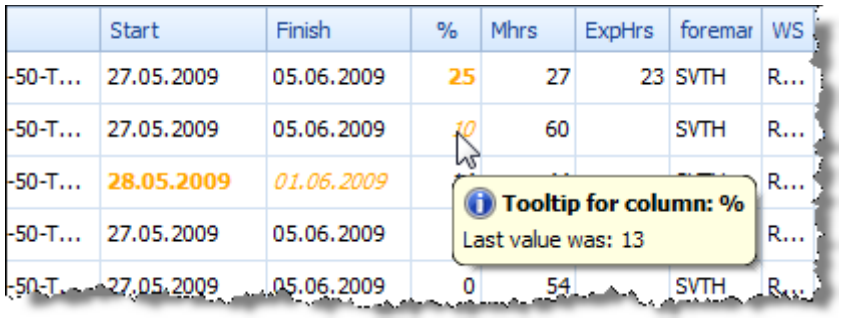
It is possible to use the markup syntax here for setting colors, font size and style.

Shadow Data

The “Shadow Data” concept in Proteus allows matching data (joined on unique key) to be added as “shadow columns” in a DataSet. This can be quite useful when comparing one set of data with another, since it is possible to have calculated columns deriving differences between them. Proteus has a feature that allows taking a snapshot of one or more DataSets, and this data is then a good candidate to use as a comparative reference point. There are two types of “shadows”, where one of them is used to monitor differences between a stored reference-point (snapshot) and the current “live” data in a DataSet. The other is more general, and allows data to be added from any kind of source as long as this source has a one-to-one relationship with the UniqueID-field for the DataSet. Each of these is described in their own subchapter here:

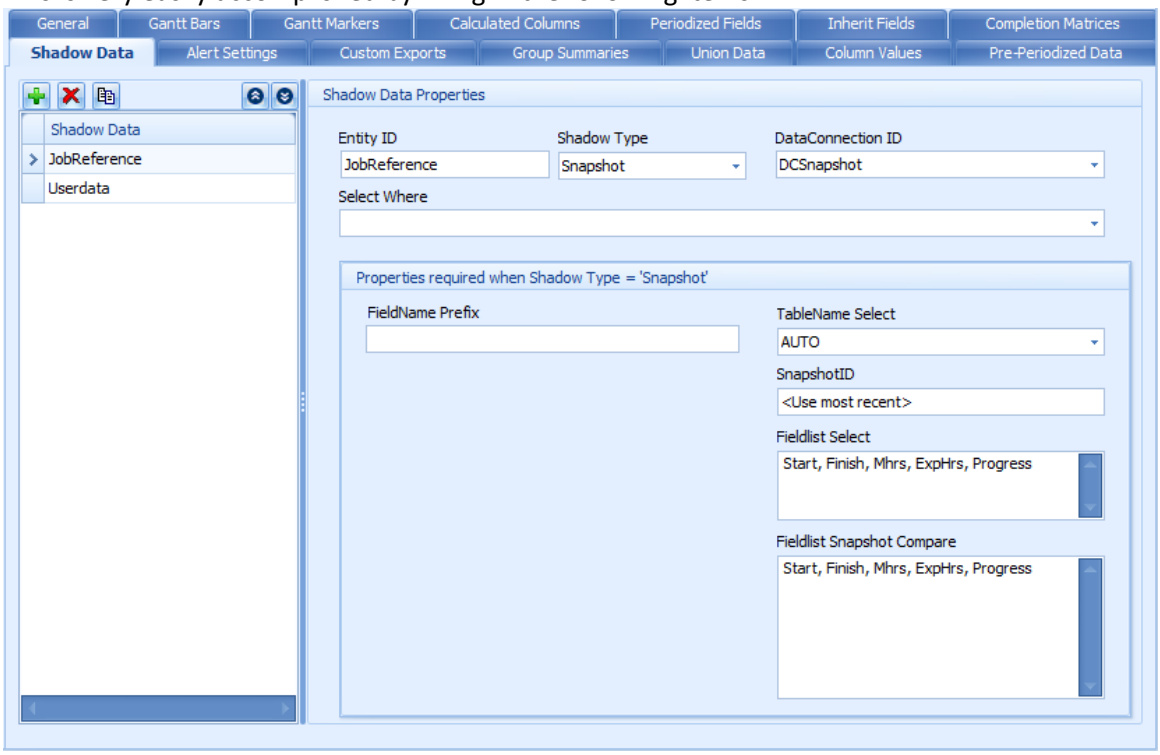
Shadow Type: “Snapshot”

As described above, it can be useful to track changes in a DataSet, relative a stored reference point. The screenshot below shows how changed values appear in orange and where a **bold** value indicates increased value, while *italic* indicate decrease. The tooltip will show the stored reference value.



A “Snapshot” shadow can highlight changes relative the snapshot.

This is very easily accomplished by filling in the following items:



The form for defining a “Snapshot” Shadow

Note: How a snapshot is set up is described in more detail in chapter on Action Buttons.

Some explanations for the various controls on screen:

Control	Description
EntityID	A unique name for the snapshot.
DataConnection ID	Pick from your list of predefined DataConnections.
Select Where	(Optional) Allows limiting the data loaded from a snapshot. This can be an Expression, i.e it can refer to an expression (that may be linked to a custom dialog): MyField = '{MyFilterValue}'
TableName Select	Type **AUTO** to automatically generate the table-names for the tables generated by a Proteus Snapshot. **Advanced technique:** The SQL that is internally generated for loading the data has this format: "**Select * from [TableName Select]**". It is possible to write a "*tablename*" that in fact is a subquery such as this: "**(Select * from MyTable where MyField=3) x**" The last "x" (*or any letter(s)*) is required in most SQL dialects as a "table alias". This may refer to an Expression using curly brackets (e.g: {MyTable})
FieldName Prefix	(Optional) Allows adding a prefix text in front of all the fieldnames provided in "**FieldList Select**".
SnapshotID	If left blank, the most recent one is used. It may sometimes be useful to refer to an expression here (e.g: {GetThisSnapshot}), where the expression (e.g with ID "GetThisSnapshot") can refer to a custom dialog that ask the user to select which of the reference points (snapshots) to use for comparison.
FieldList Select	The list of fields to select from the table (given in "**TableName Select**"). All of these fields will appear in the DataSet with the fieldname prefix "**Last_**" (unless overridden by custom prefix above).
FieldList Snapshot Compare	The list of fields to show deviations for. For each field in this list, there will be generated 3 sets of corresponding internally calculated columns: 1. "Delta_", returning the calculated difference between live and snapshot values (for dates and text this will be a Boolean indicating if they are different or not). 2. "_CellFormat_", containing the formatting code (for color and font-style) that the column will show for deviations. 3. "_ToolTip_", containing the tooltip with the "last value". See "Advanced formatting options for deviations" below if you wish to show customized formatting on deviations (instead of the orange color, using bold and italic as delta direction indicators)

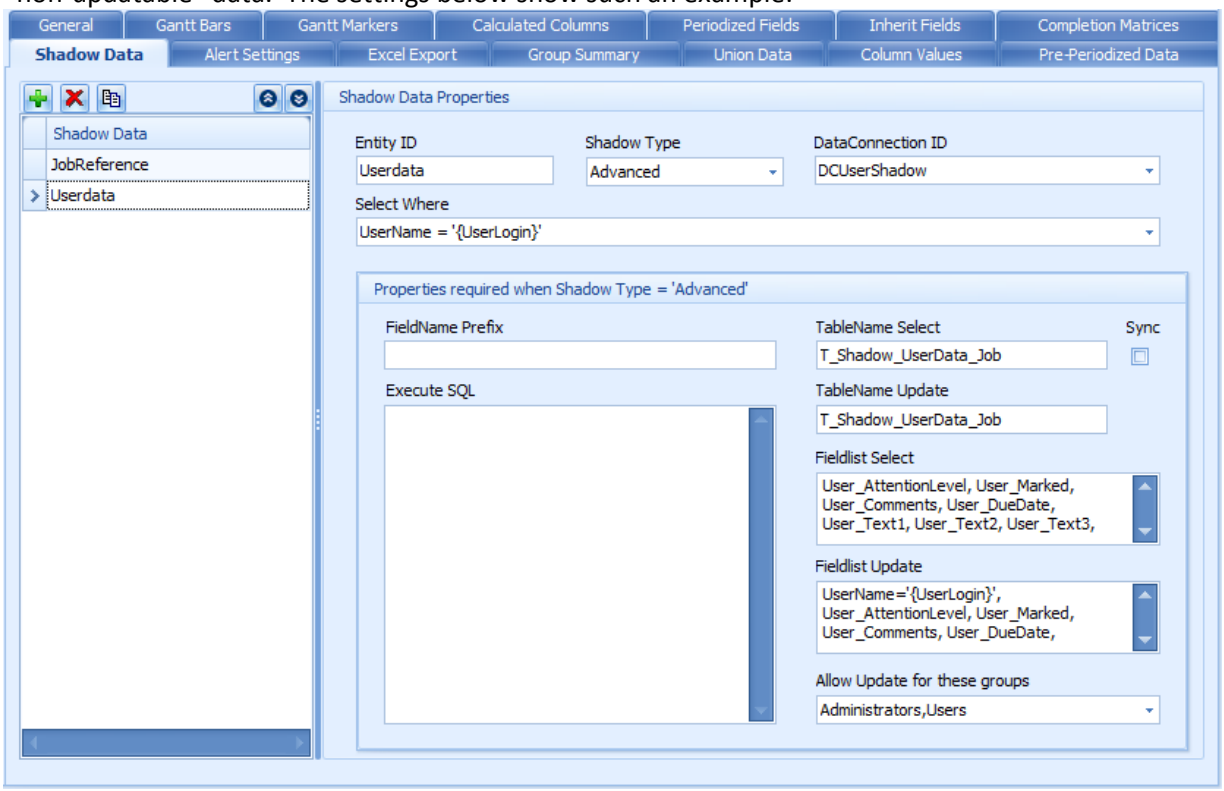
Advanced custom formatting options for compare deviations:

Each field in the "FieldList Snapshot Compare"-list can be given a customized formatting, if needed (shown in red color below):
FieldName|FontStyleIfLess§ColorIfLess|FontStyleIfGreater§ColorIfGreater

A typical Example with standard formatting: Start, Finish, Mhrs, ExpHrs, Progress
Let's say we want to show bold font and red color if greater and green, italic if smaller for column "Mhrs". This addition (in red) would achieve this result:
Start, Finish, Mhrs|Bold§red|Italic§green, ExpHrs, Progress

Shadow Type: "Advanced"

A typical use of the "Advanced" shadow is to allow "user updatable columns" to a DataSet that has "non-updatable" data. The settings below show such an example:



Note: The reference {UserLogin} above implies the existence of an expression called "UserLogin" (and this should have the expression definition "User.Login").

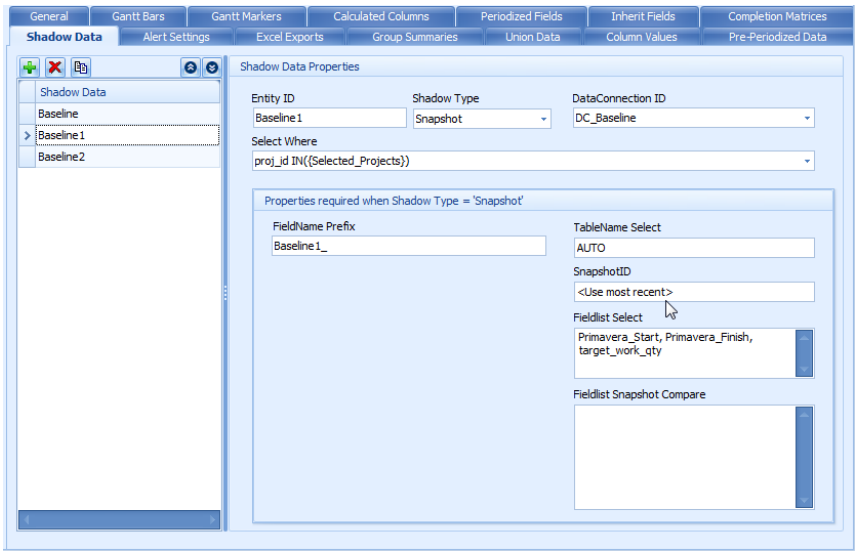
The topmost controls are identical to the ones described for "Snapshot" in previous subchapter, but the others may need some explanations:

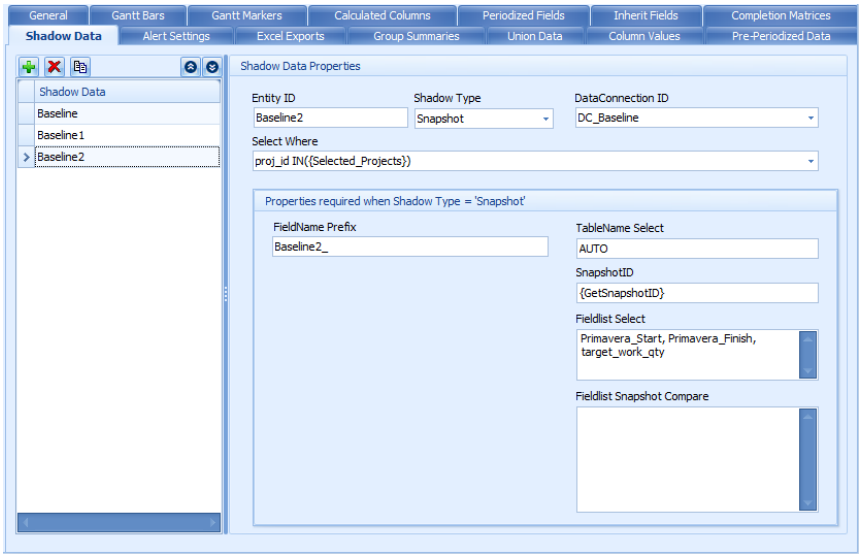
Control	Description
TableName Select	The name of the table or view to load data from. Advanced info here.
TableName Update *	The name of the table (or view) to update data to.
FieldList Select	The list of fields to select from the table (given in "TableName Select")
FieldList Update *	(Optional) The list of fields to update in the table (given in "TableName Update").
Sync	If checked, all the rows in DataSet will be written to the Shadow Table, to ensure that they are in sync. This can be useful when detecting which items have been added in DataSet since last time: The internal column "_HasShadow(N)" will be false for these added rows. It does, however, imply an initial performance hit, since potentially lots of rows have to be written back to shadow table the first time. Use with caution and only when necessary.
FieldName Prefix	(Optional) Allows adding a prefix text in front of all the fieldnames provided in "FieldList Select".
Execute SQL	(Optional) Allows running a SQL statement to modify data (normally in the table given in "TableName Select") *before* data is loaded into Proteus. *Should be used with great care!*
Allow Update for these groups	Select which Groups should be able to update the shadow columns.

*) If either of these are omitted, none of the shadow columns can be updated.

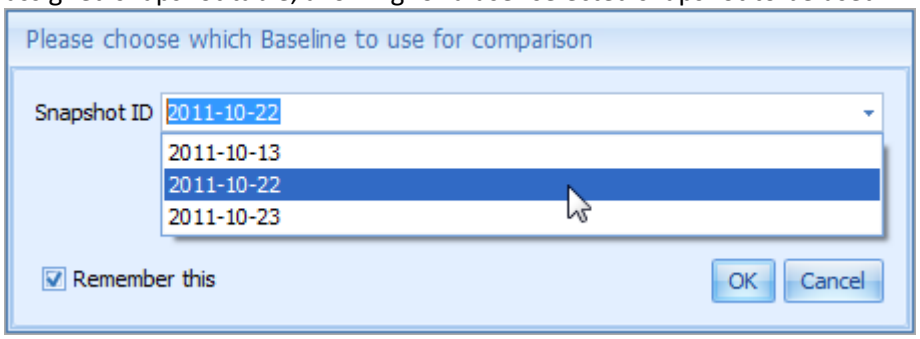
How to load more than one historical set of data from snapshot table.

In some situations you may need to compare your "live" data with not only one historical snapshot, but two or more…
The screenshots below show how 2 snapshots are virtually identical, except for the "FieldName Prefix" and "SnapshotID" settings:



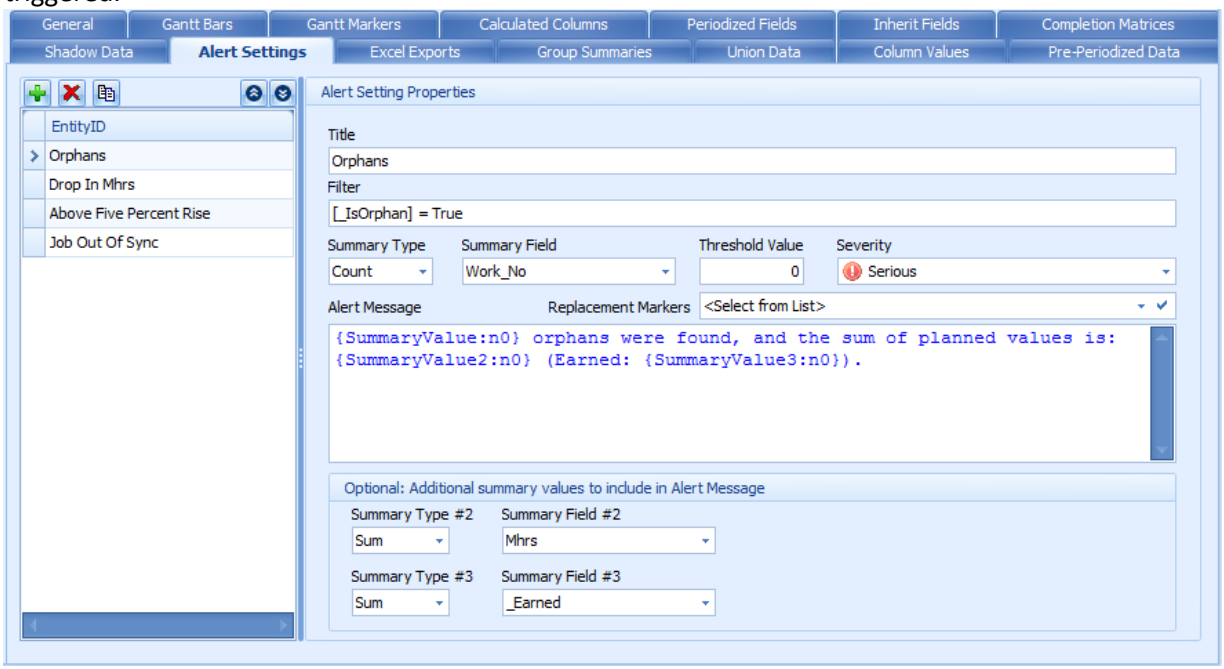


Note that the **SnapshotID** for the "Baseline2" shadow has an Expression as its setting (enclosed in curly brackets). This Expression refers to a custom dialog that lists all the snapshots present in the assigned snapshot table, allowing for a user selected snapshot to be used.



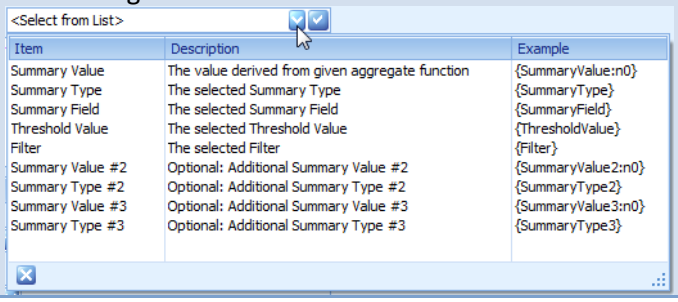
Alert Settings

In Proteus Configurator, it is possible to set up any number of "*Watchdog alerts*", and whenever a workspace is loaded at runtime, the "dog" will "bark" if any of its associated "barking criteria" are triggered.



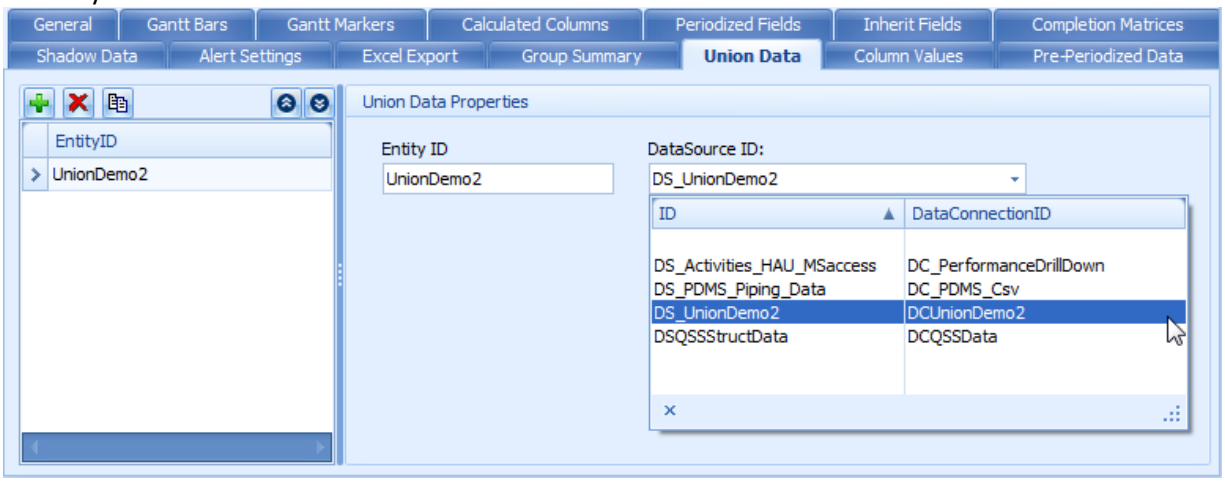
Example of an Alert Setting where the number of "Orphan" children are listed and sum of their hours are shown.

Here is an overview over what can be entered in the user interface:

Control	Description
Title	The title for the Alert.
Filter	The filter to apply to the DataSet
Summary Type	The summary operation to perform: **Count**, **Sum**, or **NotSet**. The latter implies that the Alert will be ignored.
Summary Field	(Optional) Only needs to be specified if Summary Type is "**Sum**".
Threshold Value	The threshold value for triggering the alert.
Severity	Indicates how serious it is if this alert is triggered. Choices are **Warning**, **Serious**, and **Critical**. For the last of these, a message box will be shown to ensure that the user becomes aware of the alert.
Alert Message	The text to show if alert is triggered. It is useful to have the "Replacement Markers" dropdown showing all the choices for embedding markers with text: 
Summary Type #2 and #3	As shown in example above, it can be useful to show detail information relating to an alert. In the example, the sum of "Mhrs" and "_Earned" fields are merged into the text.
Summary Field #2 and #3	See above.

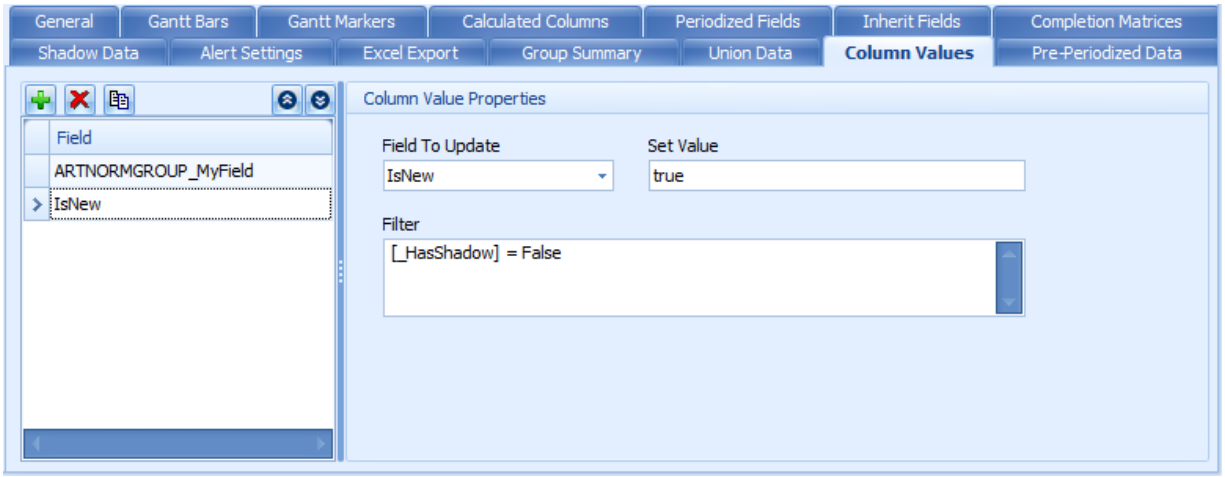
Union Data

Merging data across servers (and/or database types) is sometimes a challenge in real life. In Proteus, you can join as many DataSources together as you like, as long as the fields from each DataSource are consistent (i.e having same field names and data types). Note that the dropdown "**DataSource ID**" will only show the DataSources that have "Used in Context" set to "**UnionData**".



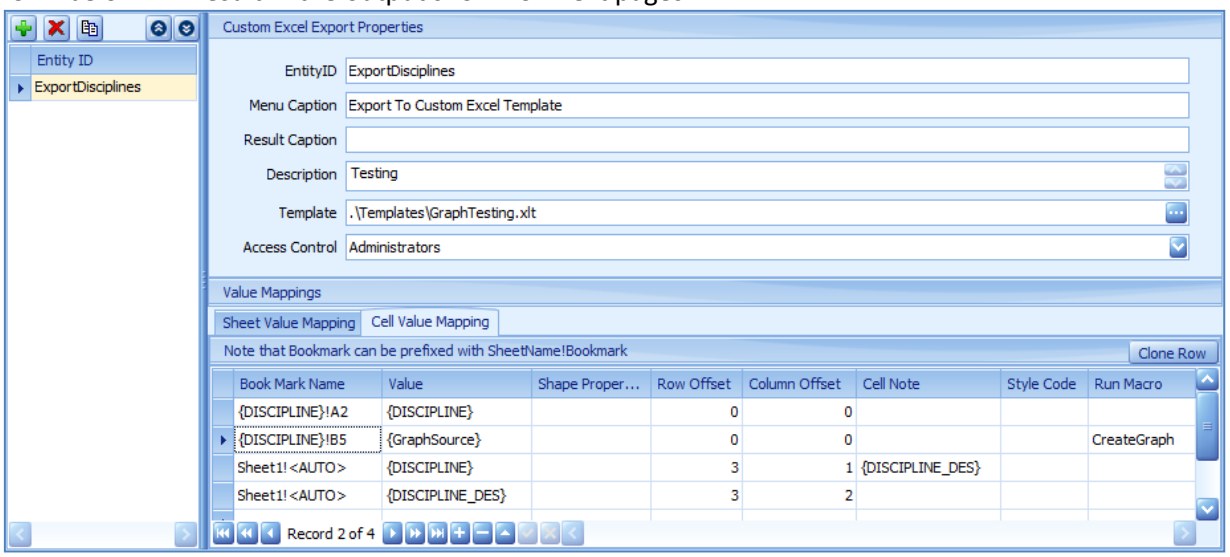
Column Values

On rare occasions, it is useful to be able to set a column value to a particular initial state when data is loaded into a DataSet. Here, it is possible to specify which field to update, for which rows, and to what value.

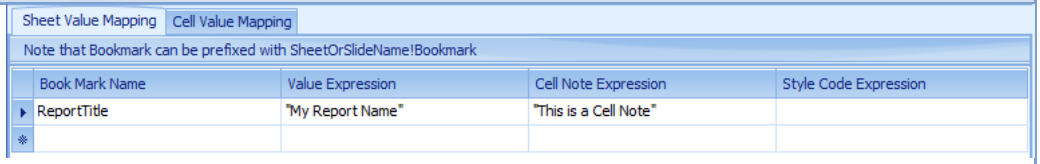
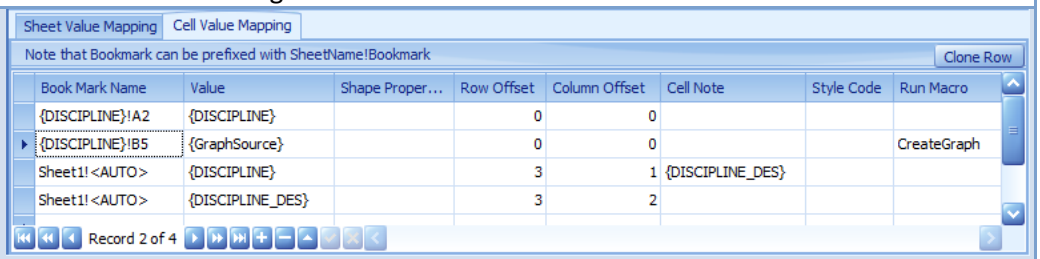


Custom Exports

It is possible to create customized **Excel** or **PowerPoint** Exports from Proteus. The settings in the form below will result in the output shown on next pages:



An example of a custom export to Excel – result will be shown and explained on next pages

Control	Description
EntityID	Give the Custom Export a unique ID.
Menu Caption	The text shown in popup menu when right clicking the Grid (with data from the DataSet for which this Custom Export is defined).
Description	For internal documentation.
Template	The path and name of Template (Excel or PowerPoint) to use. **Note**: This may be **".ppt"** (extension only) to indicate that you wish to start with a *blank* PowerPoint file (and e.g populate it with shapes). The equivalent for Excel is "**.xls**".
Access Control	Controls which group(s) can see this item when right clicking grid.
Sheet Value Mapping	 Set values at sheet level. In the above example, it sets the title of the report – this could have been an Expression (e.g: {GetTitleFromDialog}) instead of a fixed string, so that a custom dialog could have asked user what the title should be.
Cell Value Mapping	 At runtime, for each (filtered) row in the GridGantt, all of the above "actions" will be carried out. Let's use the above example (row by row) to describe what happens: 1. The value given by the field "DISCIPLINE" is set to the sheet by the same name – in cell A2. The worksheet is dynamically created if it does not exist before. 2. The value in field "GraphSource" is in a similar way set to cell B5 AND a

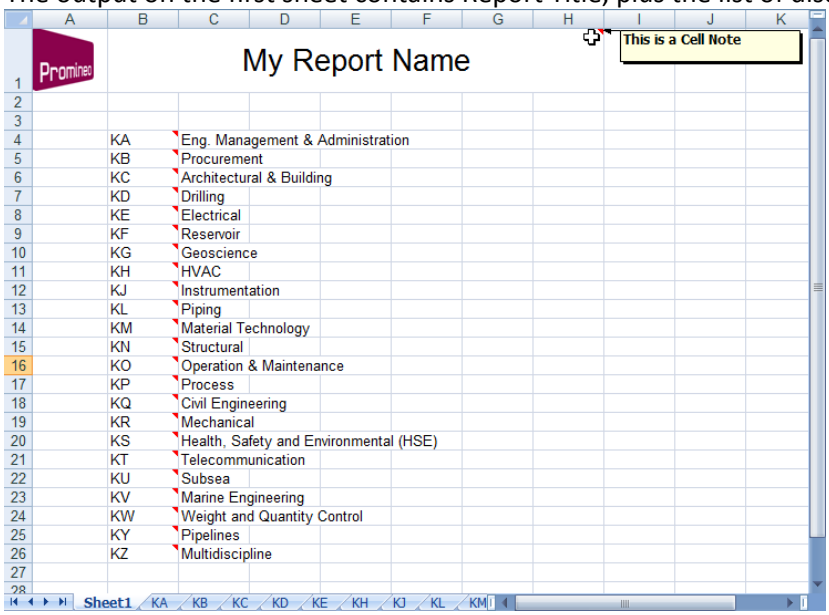
custom MACRO is triggered called “CreateGraph”. This particular macro takes the cell values from A2 and B5 and uses them to create a graph. It is important to point out that the content of the “GraphSource” field is a serialized DataTable, and in the export to Excel, the content of this data will be a corresponding range of cells.

The content of this particular custom VBA code macro is given here in small font as an example for reference:

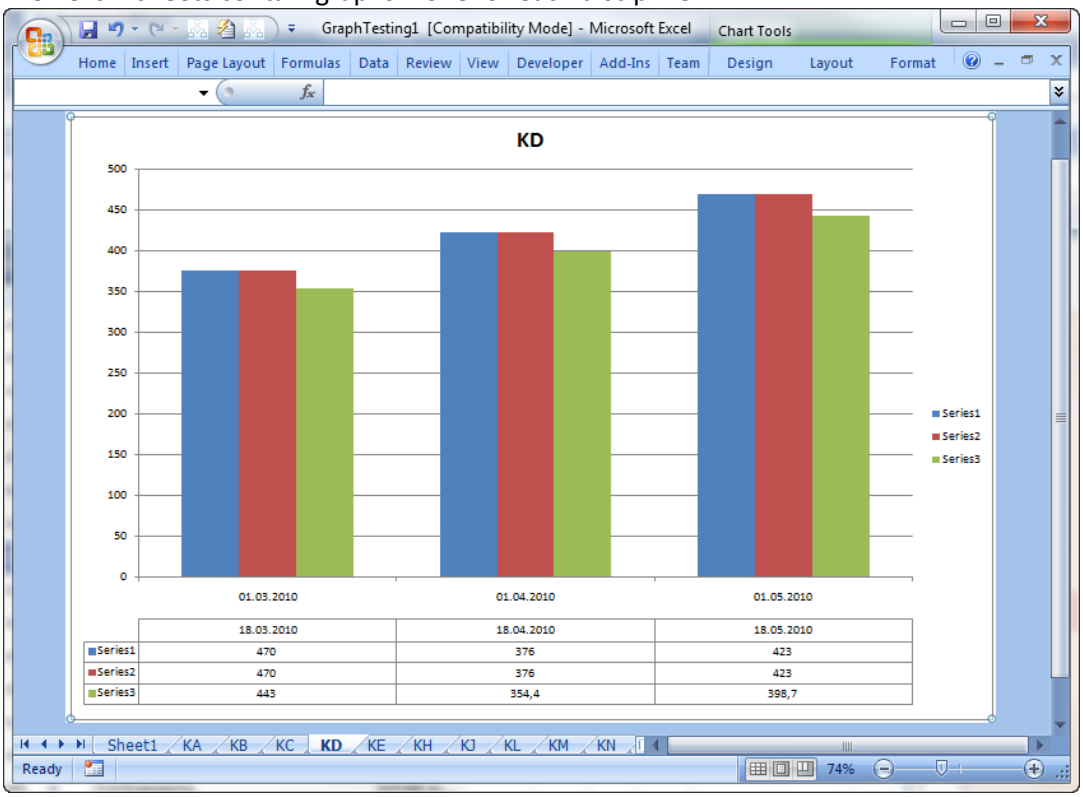
```
Public Sub CreateGraph()
  On Error GoTo ErrorHandling
  Dim vTitle As String
    vTitle = Range("A2").Value
    If vTitle = "" Then
      GoTo ExitMe
    End If
    Dim vPlotRange As Range
    Set vPlotRange = Sheets(vTitle).Range("B5").CurrentRegion
    If vPlotRange.Columns.Count <= 1 Then
      Sheets(vTitle).Visible = False
      GoTo ExitMe
    End If
    Charts.Add
    ActiveChart.ChartType = xlColumnClustered
    ActiveChart.SetSourceData Source:=vPlotRange, PlotBy:=xlColumns
    ActiveChart.Location Where:=xlLocationAsNewSheet, Name:=vTitle & " "
    With ActiveChart
       .HasTitle = True
       .ChartTitle.Characters.Text = vTitle
       .Axes(xlCategory, xlPrimary).HasTitle = False
       .Axes(xlValue, xlPrimary).HasTitle = False
    End With
    ActiveChart.HasDataTable = True
    ActiveChart.DataTable.ShowLegendKey = True
    Sheets(vTitle).Visible = False
ExitMe:
    Exit Sub

ErrorHandling:
    On Error Resume Next
    "ActiveChart.Delete
    Resume ExitMe
End Sub
```

3. The Discipline value is set to Sheet1 in a cell that starts at row 3 column 1, but then is automatically incremented (with the <AUTO> directive). Its Cell Note is set to value in Discipline_Des field.
4. The Discipline_Des field value is set one step to the right of the previously written value (in step 3 above).
5. Steps 1-4 are repeated for each row in GridGantt.

The output on the first sheet contains Report Title, plus the list of disciplines and descriptions.



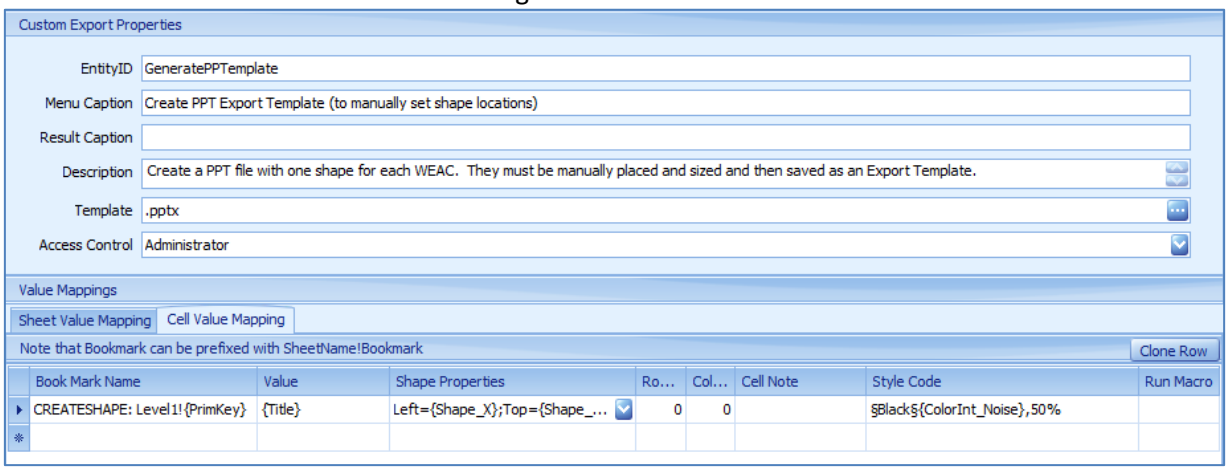
The next N sheets contain graphs – one for each discipline:



Sheets are dynamically generated and a custom macro creates charts for each one.

Example of generating an Export template:

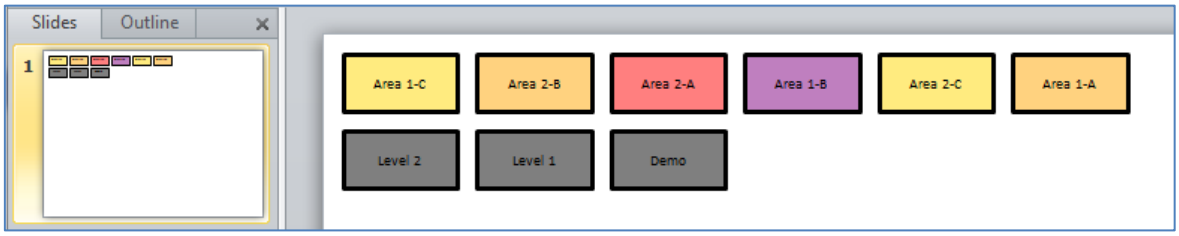
It is possible to create shapes in an Excel or PowerPoint document, by adding "**CREATESHAPE:**" in front of the bookmark name. Similarly, you can set the properties of an existing shape (in a template) by adding "**SHAPE:**" instead. The Style Code will in both cases control the properties of the shape. Below is a screenshot where shapes are generated in a blank PowerPoint presentation, and will be used as a "Noise Map", where each area of an offshore rig gets a color according to the measured noise level. Let's look at some of the settings:



Column Name	Value	Description
Bookmark Name	CREATESHAPE: Level1!{PrimKey}	Create the shape (or use the shape with given bookmark, if it exists) in slide named "Level1" (will

		be created if not found), and sets its name to the content of field "PrimKey". Note: It is possible to add a parameter to the CREATESHAPE syntax, in 2 distinct ways: 1. CREATESHAPE(<N>). Here "N" is a number that corresponds to the shapetype in Office. See office VBA documentation for more info. 2. CRETESHAPE(<CoordinateList>). Here is an example to clarify the "CoordinateList" syntax: Polygon: {[16.2,30.6], [11.0,30.6], [11.0,50.8], [16.2,50.8]}. This is the same format that Proteus uses to represent polygons in the DataLayout's Overlay control (see Appendix for more info).
Value	{Title}	Set the text in the shape from "Title" field.
Style Code	§Black§{ColorInt_Noise},50%	This sets some color properties of the shape. See format definition here.
Shape Properties	AUTOARRANGE=60;30;6;10	Here, any shape property can be set. Typical example would be "Left={MyNumericField}". **Note**: When using the "**CREATESHAPE**" syntax in bookmark, it is often desired to have the shapes arranged in some way. This can be achieved by the special syntax: **AUTOARRANGE**=Width;Height;NoPerRow;Margin

The resulting PowerPoint document is shown below. Each shape must then be manually sized and modified to match the geometry of the area on the rig (the plot-plan).



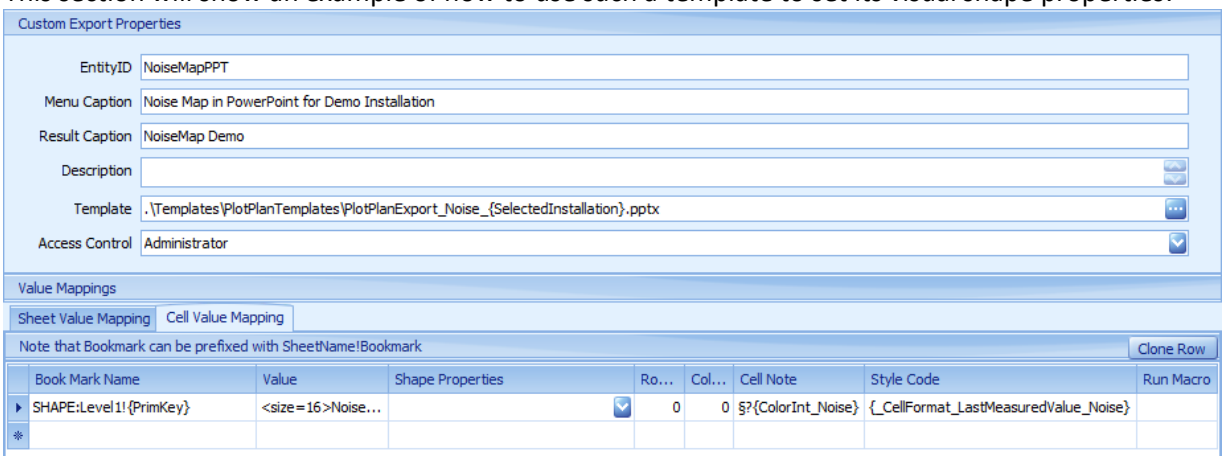
When this process is done it must be saved as a pptx (or ppt) file and can then be used as a Template for setting shape properties. In such a way the shape colors can be updated from Proteus by the click of a button, at any time, according to the current noise measurements.
How this is set up in Configurator is shown in next section.

These steps may be useful when generating a multi-slide template:

1. Ensure that each object has a correct slide name (the text used in front of "!" in bookmark).
2. Create background bitmaps for each of the slides and place them in the subfolder "GUIImages".
3. For each object, define the name of the associated bitmap (without path). Let's assume we use a field for this purpose called "OverlayBitmap".
4. Use the AutoArrange setting in Shape Properties to produce shapes for each slide
5. Create a DataLayout with an OverLay Control where the "OverlayBitmap" field is used to select image.
6. Edit the Shapes in Overlay Control, and run the custom export. Save as template.

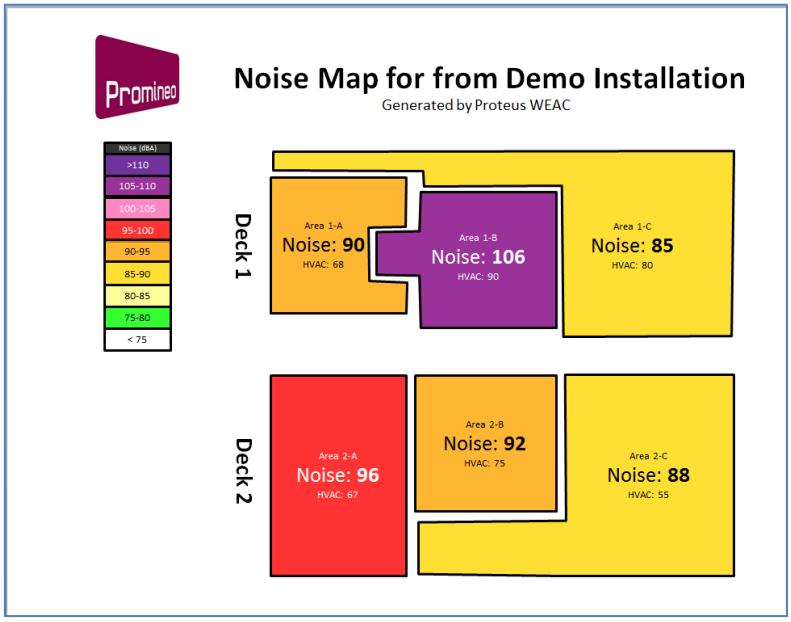
Example of setting shape properties for an existing Export template:

The previous section demonstrated how an Export Template could be created through a Custom Export, and how the shapes had to be manually modified to their proper shapes, sizes, and location. This section will show an example of how to use such a template to set its visual shape properties:



Column Name	Value	Description
Bookmark Name	SHAPE: Level1!{PrimKey}	Reference the shape in slide named "Level1", with shape name given by "PrimKey" field.
Value	WEAC: {Title} <size=16>Noise: {LastMeasuredValue_Noise}</size> HVAC: {LastMeasuredValue_Noise_HVAC}	The text including markup to show in the shape.
Style Code	{_CellFormat_LastMeasuredValue_Noise}	This sets some color properties (including transparency) given by a calculated expression.
Cell Note	§?{ColorInt_Noise}	When bookmark points to a shape, the Cell Note column can be used to set the foreground properties – in this case it prefixed the background color with a "?" sign to return a "readable" color (to avoid e.g black color on black background!) More on color syntax.

The resulting PowerPoint document may look like this:



Chapter 8: Creating a dynamic interface – The Proteus DataLayout

The idea behind the Proteus DataLayout is to "wire up" an interface to any existing (hierarchically structured) data. This chapter will give an insight into how this is accomplished.

Here is a screenshot of a demo Workspace that shows three hierarchical sets of data – Site, Network, and Discipline. Each selected item will show a KPI (in this case Productivity) for each of the 3 levels, and the discipline level shows an additional KPI as well as a state indicator (traffic light) and a graph showing the history of the discipline (derived from successive snapshots from Proteus).
The browser control in the upper right corner will show the Google Map location for the selected Site.

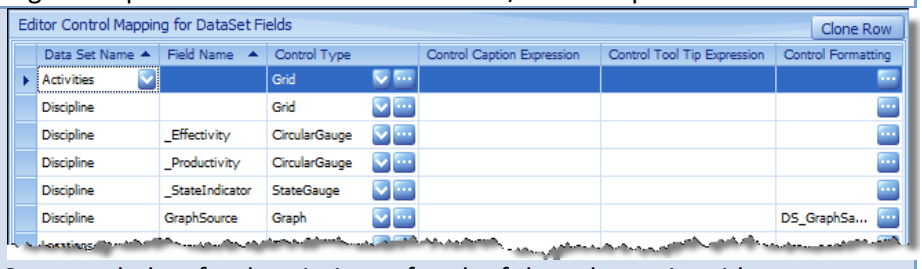
A demo example of controls (grids, gauges, graphs, browser control, etc) arranged in a Proteus DataLayout.

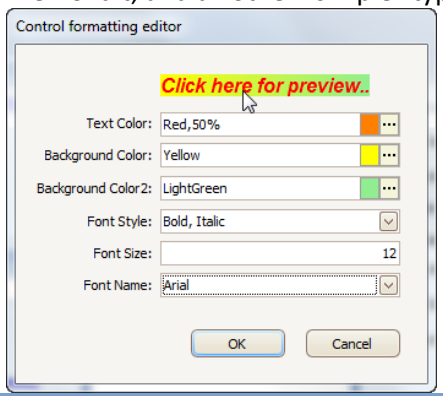
The following type of elements can be used (dragged and dropped) in a DataLayout:

LookupEdit	Multicolumn Dropdown
ComboBox	Dropdown
CheckedComboEdit	Dropdown Checklist
ImageComboBox	Image Dropdown
CheckBox	Checkbox (in various visual styles)
DateEdit	Date Picker control
TextEdit	Textbox (single line)
MemoEdit	Multiline Textbox
MemoExEdit	Multiline Popup Textbox
ButtonEdit	Textbox with button (for browsing for files etc)
Hyperlink	Hyperlink (with optional browse button)
SpinEdit	Numeric Spin-control
TrackBar	Slider control
PictureEdit	Picture control
Label	Label control
TreeList	Tree List (showing hierarchical tree structure). More info here.
The controls above this row are also available for Custom Dialogs, but not the ones below:	
Grid	Grid Control
CircularGauge	Gauge with circular scale

LinearGauge	Gauge with linear scale
StateGauge	Traffic light style indicator
Graph	Generic graph component
Browser	Browser control
Overlay	Allows having (editable) shapes overlaying a bitmap. More info here.
RichTextEdit	RichText Edit control
TextProcessor	A full scale Text Processor (resembling Microsoft Word)

Here is the list of controls in the user interface (see screenshot of form on next page):

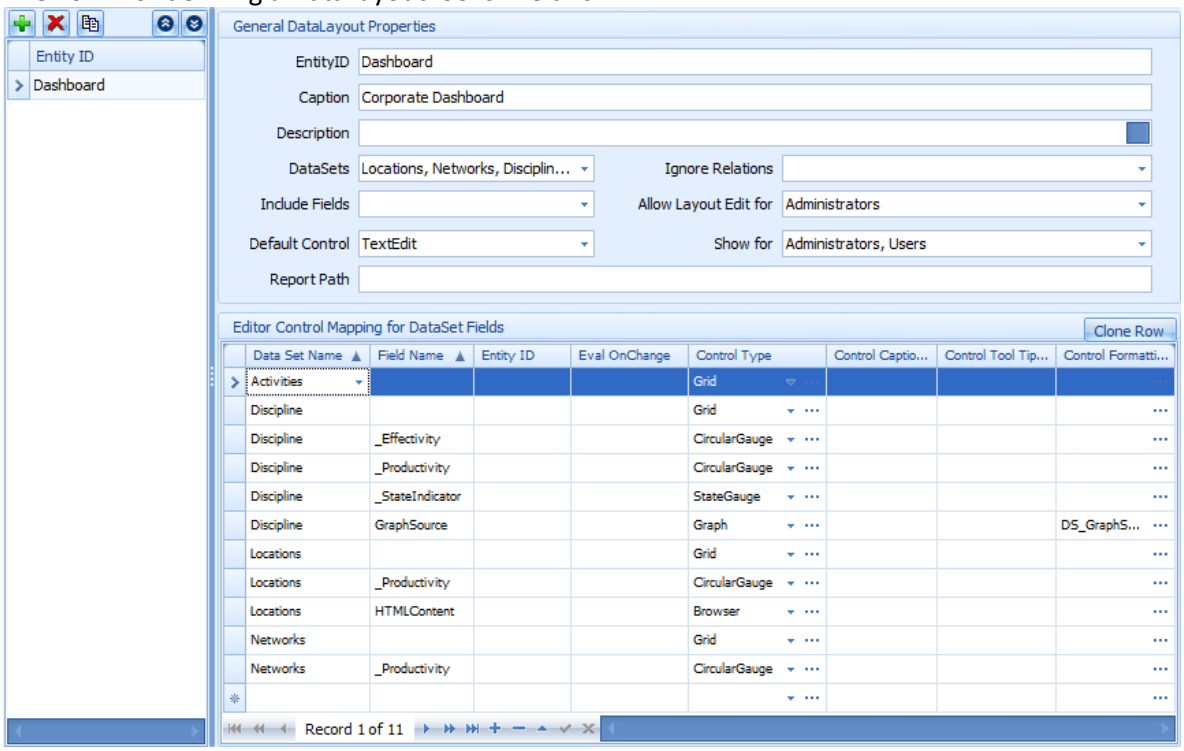
Control	Description
EntityID	The unique ID for this DataLayout
Caption	The caption on the tab shown in Proteus
Description	Used for internal documentation
DataSets	The list of DataSets to generate default controls for
Default Control	The default type of control to use (TextBox or MemoEdit)
Include Fields	Control which types of fields to automatically create controls for. It is recommended to set this to "**none**" for performance reasons, so that the only controls that are created are the ones explicitly specified in "Editor Control Mappings for DataSet Fields"-grid (see few rows down).
Ignore Relations	(Optional) Allows for breaking up hierarchical binding(s) between DataSets, so that they may act independently of another.
Allow Layout Edit for	Controls which Groups can edit the layout (right click).
Show for	Controls which Groups can see this DataLayout tab.
Report Path	(Optional) The reports found at this path will appear in a dropdown at runtime in the DataLayout's top right corner. **Note**: You are recommended to avoid this setting, and instead use the regular Report Wizard in Proteus to create/launch reports.
Editor Control Mapping for DataSet Fields	 See rows below for descriptions of each of the columns in grid:
Columns in grid for customizing controls in a DataLayout:	
DataSet Name	Select which DataSet the control is intended for
Field Name *	Select which field in the above DataSet to bind to the control.
EntityID	This needs to be set ONLY if the control is used in unbound mode (having a fieldname in angular brackets). More info here.
Eval OnChange	This can be set to the expression to be evaluated when the control's value changes. Useful if you want e.g a slider control's value to be reflected in a textbox, as it is being dragged.
Control Type	Select which control type to use (see list above this table). The ellipsis may bring up a wizard for building setting properties for the various controls. Not all control types will allow opening this wizard, e.g Grid – it has its own runtime options for setting properties that will be stored along with the layout data.

Control Caption Expression	(Optional) Allows setting control caption from expression (possibly with a lookup from a database).
Control ToolTip Expression	See above.
Control Formatting	Press the ellipse button to bring up a form for setting control background and foreground colors as well as font properties for a control (such as MemoEdit, and all other "simple" types): 

***) Notes on FieldName naming conventions:**

1. It is possible for controls to be "*unbound*" (i.e not related to any particular source field), and in this case an ad hoc "fieldname" can be created by enclosing a text in angular brackets (E.g: "<MyID>"). For more info on such unbound fields, see section "Working with unbound controls in a DataLayout".
2. For Graph control, it is also possible to use an expression (in curly brackets), or typing "**<DATASET>MyID**" to use the DataSet as Graph's source. If you have more than one Graph for a DataSet, you need to have unique names following the <DATASET> marker.
3. If you have more than one Grid, TreeList , PivotChart, or Overlay for the *SAME DataSet*, then type a name in this column to make it unique (E.g: "Grid1").

The Form for defining a DataLayout looks like this:

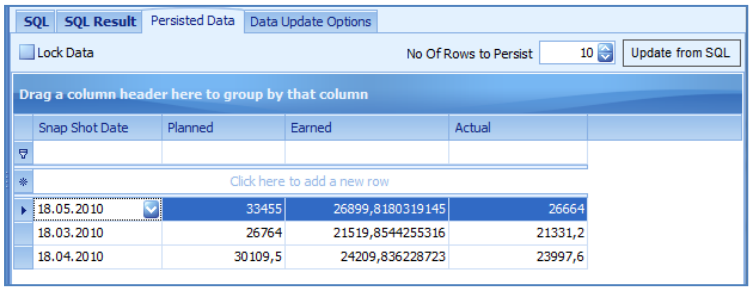


Some useful hints on working with Graphs in a DataLayout:

When setting up a graph source it is useful to follow these steps (the example can be found in the "Performance Drilldown" Workspace in "Proteus Demo Repository"):

Step1: Create a DataSource for Sample Data.

First make a "dummy" DataSource to get some sample data to use when running the Graph's Design Wizard. It is important to *persist* this data so that it is available when running Designer later.

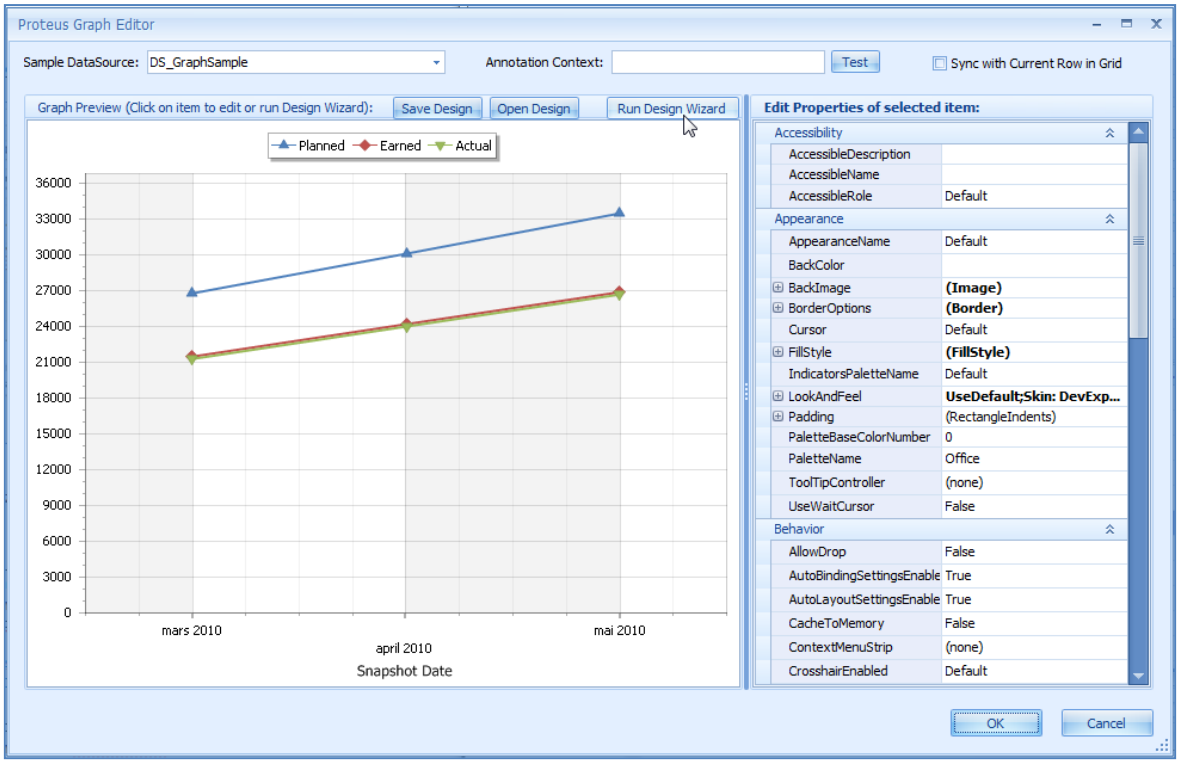


The SQL may look like this:

```
SELECT SnapShotDate,
[_SumChildPlanned] AS Planned,
[_SumChildEarned] AS Earned,
[_SumChildActual] AS Actual
FROM Discipline WHERE
GLOBAL_DISCIPLINEID='2005-1'
```

Note how this SQL is reused later in the calculated column (next page).

Step2: Add a Graph control in DataLayout and press ellipsis button (the three dots).



Then select the DataSource in the dropdown in upper left corner.
Press the "Run Designer Wizard" button to go through the steps of setting the multitude of properties that a Graph can have.

Step3: Create a Calculated Column(e.g: "GraphSource") to hold the source data for the graph.

You need to give this column a somewhat complex looking expression that converts (serializes) an entire DataTable into a string format:

```
cn#MyConnection.GetDataTableSerialized("SELECT SnapShotDate,
[_SumChildPlanned] AS Planned, [_SumChildEarned] AS Earned,
[_SumChildActual] AS Actual FROM Discipline WHERE
GLOBAL_DISCIPLINEID='" + GLOBAL_DISCIPLINEID + "'")
```

Note: When testing such a calculated column (by pressing the “Test” button), if it is correct you should see the result starting with “<NewDataSet>…” – an xml structure representing the returned records.

Note2: It is possible to have an expression be used directly as source, instead of using a field value. Simply supply the name of the Expression in curly brackets as the “Field Name”.

Note3: For a Graph, you may (as mentioned earlier) use the DataSet as source by typing "**<DATASET>**MyID" as the "Field Name".

Step4: Set the Field Name property for the Graph to “GraphSource” in DataLayout editor.
This concludes the work with setting up the Graph in Proteus Configurator, now only remains to drag it onto the DataLayout at runtime (in Proteus) and give it a proper size and location relative to the other controls – and then save the layout to Public [Default], the one that will be brought up at startup.

Controlling Chart details using special calculated column names:

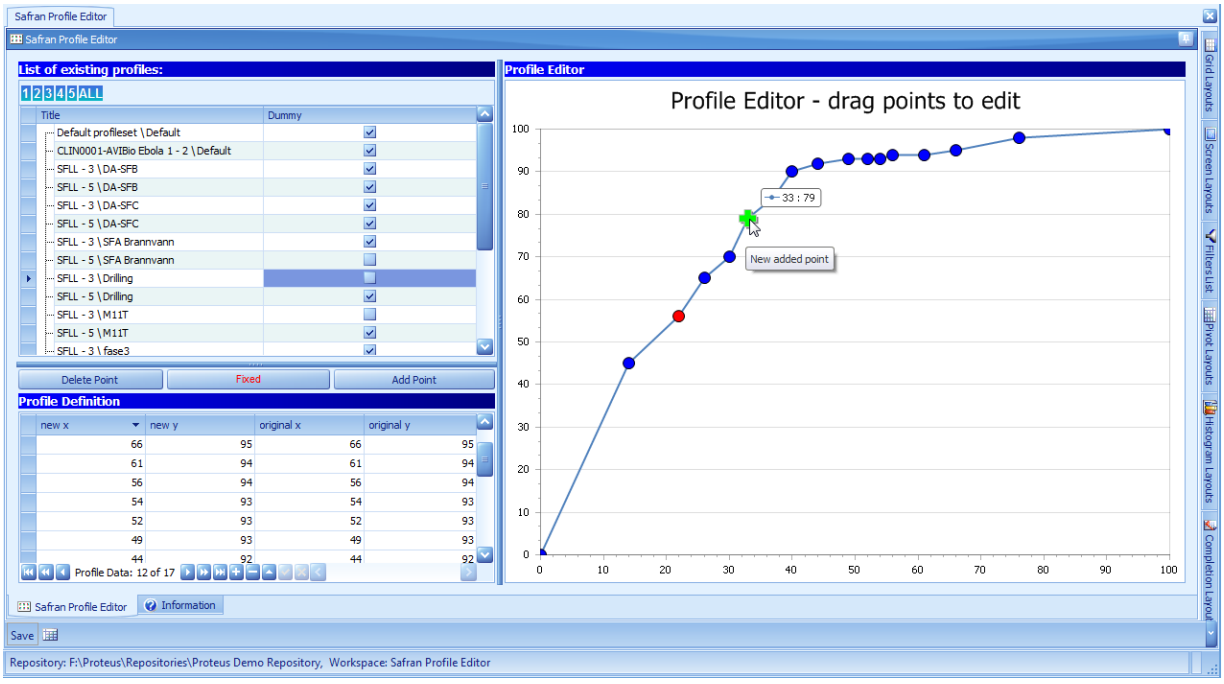
Here is the list of special (reserved) names to use if you wish to control visual aspects of a chart. The fieldnames all start with `"_CustomDrawSeriesPoint_":`

Control	Description
`LabelText`	Control the label text
`Color`	The color of the point
`Color2`	The gradient color (if gradient – see below) of the point
`BorderColor`	The color of the point's border
`ShadowColor`	The color of the point's shadow
`ShadowSize`	The shadow size
`MarkerKind`	The name of the marker (see previous section on Gantt Marker)
`MarkerSize`	The size of the marker
`ToolTip`	The tooltip format string to use for creating tooltip content (see Calculated column _ToolTip_FieldName).
`AllowEdit`	If true, then point can be dragged by user
`ValueExpression1`	The expression text that controls how the series value should be generated that is written back to source.
`ValueExpression2`	See above, but used for argument value.
`FillMode`	Values are: **Gradient, Solid, Hatch**
`GradientMode`	Must be set according to "**FillMode**". E.g: If Gradient then "LeftToRight".

Note: The above fields can be suffixed with a particular SeriesName in case you have multiple chart controls in a DataLayout. You can also append **"_Selected"** to fieldname if you wish to control properties for selected item in chart.

Control	Description
`CustomDrawAxisLabel_AppearanceExpressionX`	X-Axis label appearance
`CustomDrawAxisLabel_AppearanceExpressionY`	Y-Axis label appearance

Example of how to make an interactive custom Profile Editor for Safran:



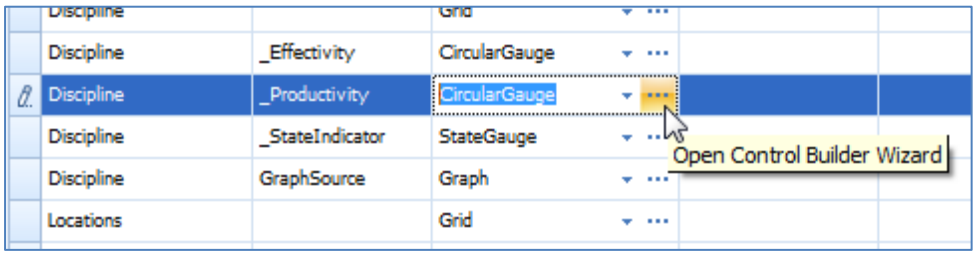
The screenshot above shows a DataLayout where the left-topmost grid lists all available profiles (from Safran database), and the left-lowermost grid show the coordinates for the selected profile.
The following calculated columns in the DataSet relating to the profile coordinates ("Profile Data") results in the interactive functionality whereby a coordinate in chart point can be "dragged", and its visual properties controlled:

ColumnName	DataType	Expression
_CustomDrawSeriesPoint_AllowEdit	System.Boolean	true
_CustomDrawSeriesPoint_BorderColor	System.String	'black'
_CustomDrawSeriesPoint_BorderColor_Selected	System.String	'silver'
_CustomDrawSeriesPoint_Color	System.String	Iif(orig_value_x <> value_x or orig_value_y <> value_y, 'red', iif(UniqueId='New','Lime', 'blue'))
_CustomDrawSeriesPoint_LabelText	System.String	'(' + value_x + ',' + value_y + ')'
_CustomDrawSeriesPoint_MarkerKind	System.String	Iif(UniqueId = 'New','plus','circle')
_CustomDrawSeriesPoint_MarkerSize	System.Int32	12
_CustomDrawSeriesPoint_MarkerSize_Selected	System.Int32	18
_CustomDrawSeriesPoint_ShadowColor	System.String	'transparent'
_CustomDrawSeriesPoint_ShadowColor_Selected	System.String	'gray'
_CustomDrawSeriesPoint_ShadowSize	System.Int32	2
_CustomDrawSeriesPoint_ToolTip	System.String	Iif(UniqueId = 'New', 'New added point','Original point: (' + orig_value_x + ',' + orig_value_y + ')')
_CustomDrawSeriesPoint_ValueExpression1	System.String	'General.Iif(Value < 0, 0, General.Iif(Value > 100, 100, General.Iif(Value < MainDataSet.Tables["Profile Data"].LookupValue("value_y DESC", "Id ="{Id}" and value_x<{value_x}"), MainDataSet.Tables["Profile Data"].LookupValue("value_y DESC", "Id ="{Id}" and value_x<{value_x}"), General.Iif(Value > MainDataSet.Tables["Profile Data"].LookupValue("value_y ASC", "Id ="{Id}" and value_x>{value_x}"), MainDataSet.Tables["Profile Data"].LookupValue("value_y ASC", "Id ="{Id}" and value_x>{value_x}"), Convert.ChangeType(Value , Convert.GetTypeCode(0))))))'
_CustomDrawSeriesPoint_ValueExpression2	System.String	Iif(UniqueId<>'New','General.Iif(GetFixedMode, Convert.ChangeType(Value , Convert.GetTypeCode(0)), orig_value_x)', 'General.Iif(Value < 0, 0, General.Iif(Value > 100, 100, Convert.ChangeType(Value , Convert.GetTypeCode(0))))')

The above example is taken from a Workspace called "Safran Profile Editor" in the repository "Proteus Demo Repository" (available on request from Promineo).

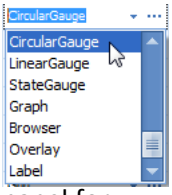
Some useful hints on defining Gauge properties:

There is quite an extensive set of properties to set for a gauge, and below are some (hopefully) useful hints for getting started:

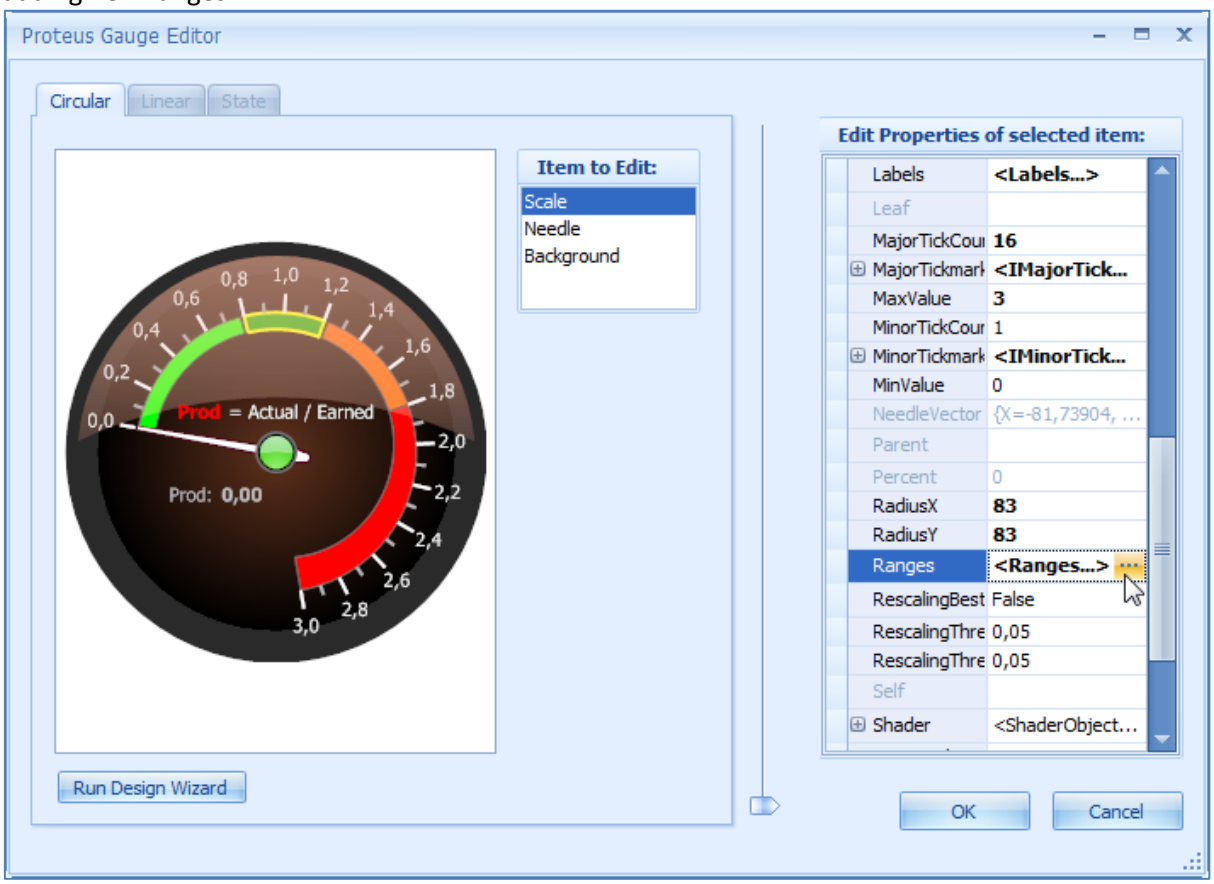


Select the CircularGauge control from list (press the cell right of it first) and then the "..."

In a DataLayout, choose which (KPI) field you wish to see in a gauge – this will normally be a calculated field.



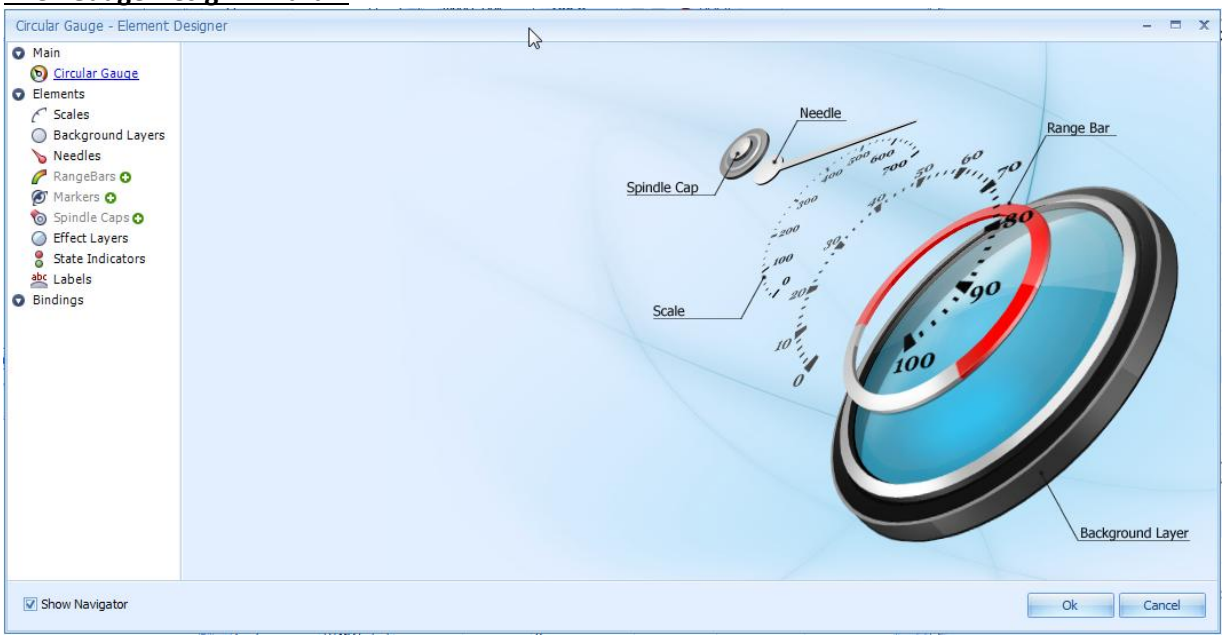
Select which type of gauge you wish to use (see the 3 topmost items in list at right), and press the ellipsis (...) button.
You will then see a preview of what the gauge looks like, and can set various properties relating to Scale, Needle, and Background in the right hand property grid.
One of the most typical tasks is to set the colored interval ranges for the circular or linear gauges. The property called "**Ranges**" has an ellipsis (...) button that opens up a panel for adding new ranges:



The editor form that is seen when "..."is pressed. Note the "Run Design Wizard" button (more on next page)

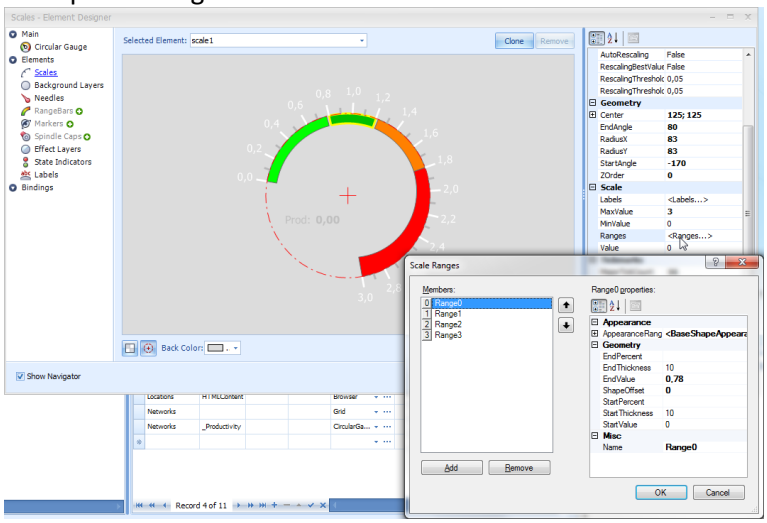
It is possible to set virtually all the properties for the gauge in the right property grid, but a more convenient way is to press the "Run Design Wizard" and be guided though various options. More on this on next page.

The "Gauge Design Wizard":

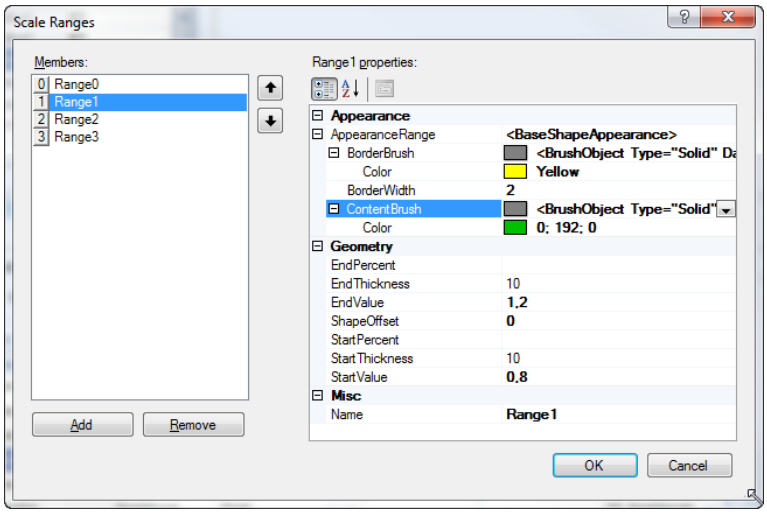


The various categories of properties to define for a gauge is shown in list a left, and each will bring up a new form (see below).

Example: Setting Scales



One of the most common properties for scales is to set "Ranges" see below.



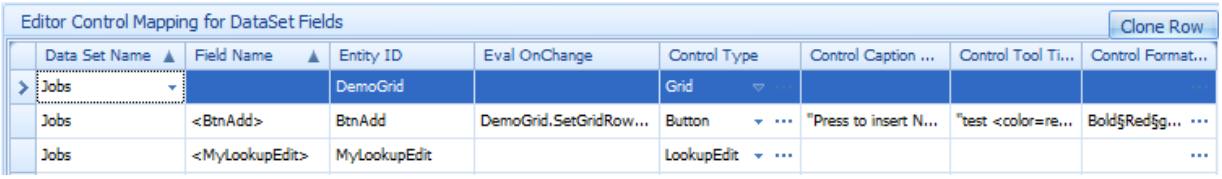
Each range's properties can be set using this form.

Working with unbound controls in a DataLayout

We will use a specific example here to illustrate the principles of using unbound controls, and buttons in a DataLayout interface. The example is taken from Workspace "Interactive DataLayout Demo" in the "Proteus Demo Repository":

Let's assume we want a multi column dropdown (this control is called a "LookupEdit" in Proteus), and after selecting an item in this list, a button can be pressed that adds a new row to the grid and populates the first and second columns with values from selected dropdown's first 2 columns.

In Configurator we need 3 items (controls) to achieve this:

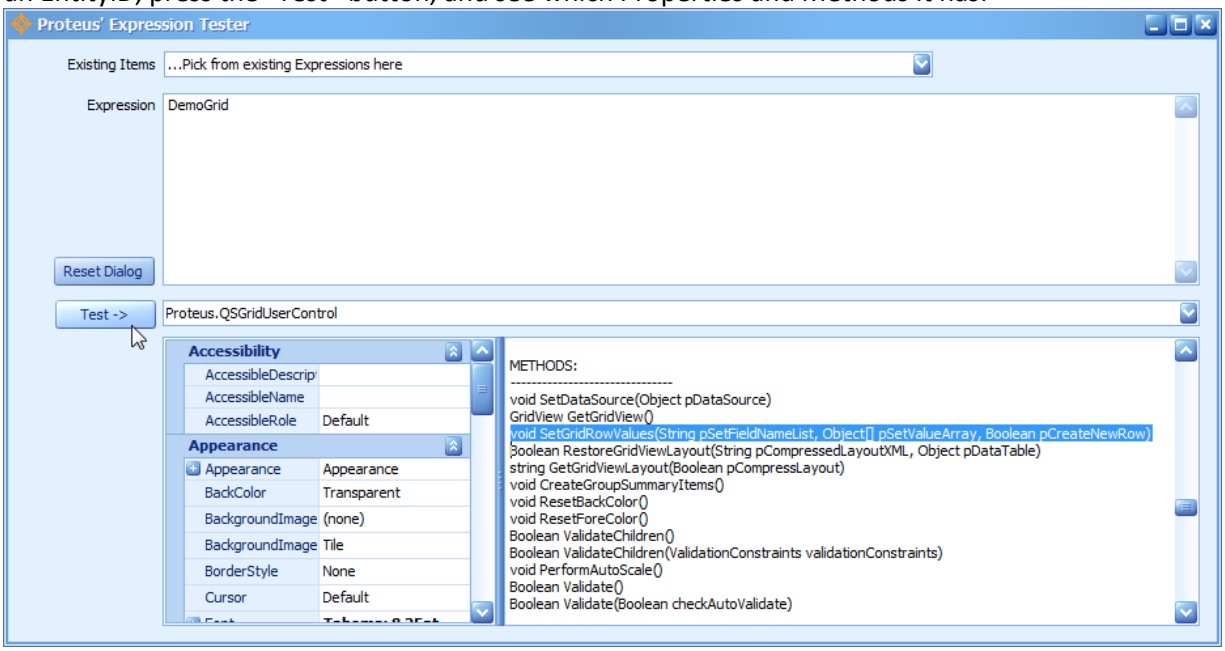


Notice that all of the 3 controls (Grid, Button, and LookupEdit) all are given an EntityID. The Grid (given the EntityID "DemoGrid") is bound to the "Jobs" DataSet, and there is a method we can call (or more precisely: An Expression we can evaluate) that will insert a new row in the grid, and set its column values (colors are used to indicate distinct parts of the Expression):

```
DemoGrid.SetGridRowValues(
"Work_No, Activity_No",
{ DataLayout.GetControlValue(MyLookupEdit,0),
DataLayout.GetControlValue(MyLookupEdit,1) },
True)
```

Rhetorical Question: How do we know that the DemoGrid has a method called "SetGridRowValues" with 3 parameters (blue, purple and orange texts) – Is this something we "just have to know"?

Answer: If we start Proteus (and thereby create the 3 controls – in Configurator they are not created, just defined) and open the "Expression Tester" from the Tools menu, we can simply type the name of an EntityID, press the "Test"-button, and see which Properties and Methods it has:



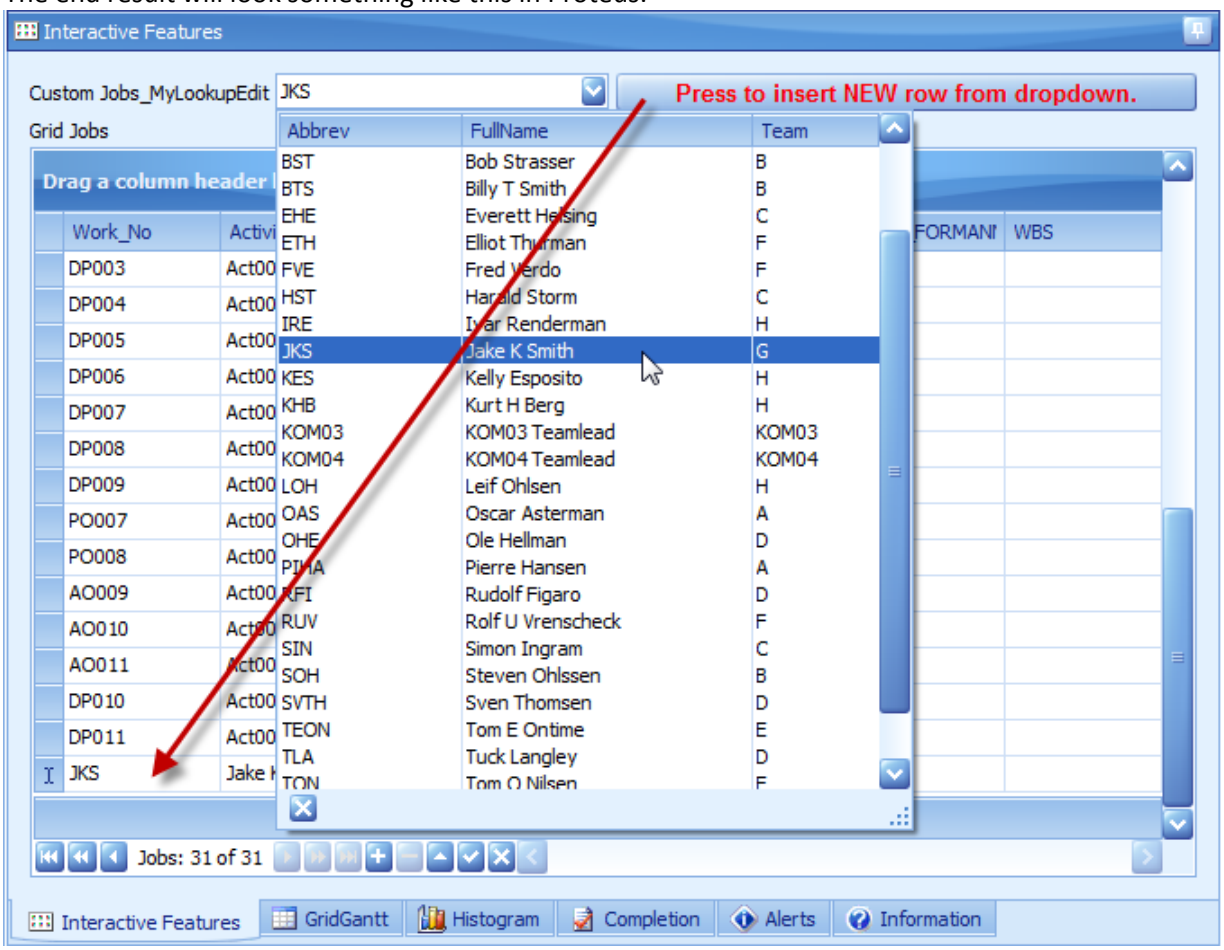
In Proteus we can launch the Expression Tester to get more info on various objects.

On previous page, the highlighted text shows the name of the method and the parameters it requires:
void SetGridRowValues(String pSetFieldNameList, Object[] pSetValueArray, Boolean pCreateNewRow)

We will not dwell too much on the specific details of this method here, but simply accept that it exists, and that it can be supplied with a string of fieldnames, an array of values (in curly brackets), and a boolean (binary value) indicating if we should create a new row or not.

Once this is established, we need to have a button that, when pressed, will evaluate the above expression. As seen in Configurator screenshot above (on the second of the three rows), the "Eval OnChange" setting has the mentioned Expression text.

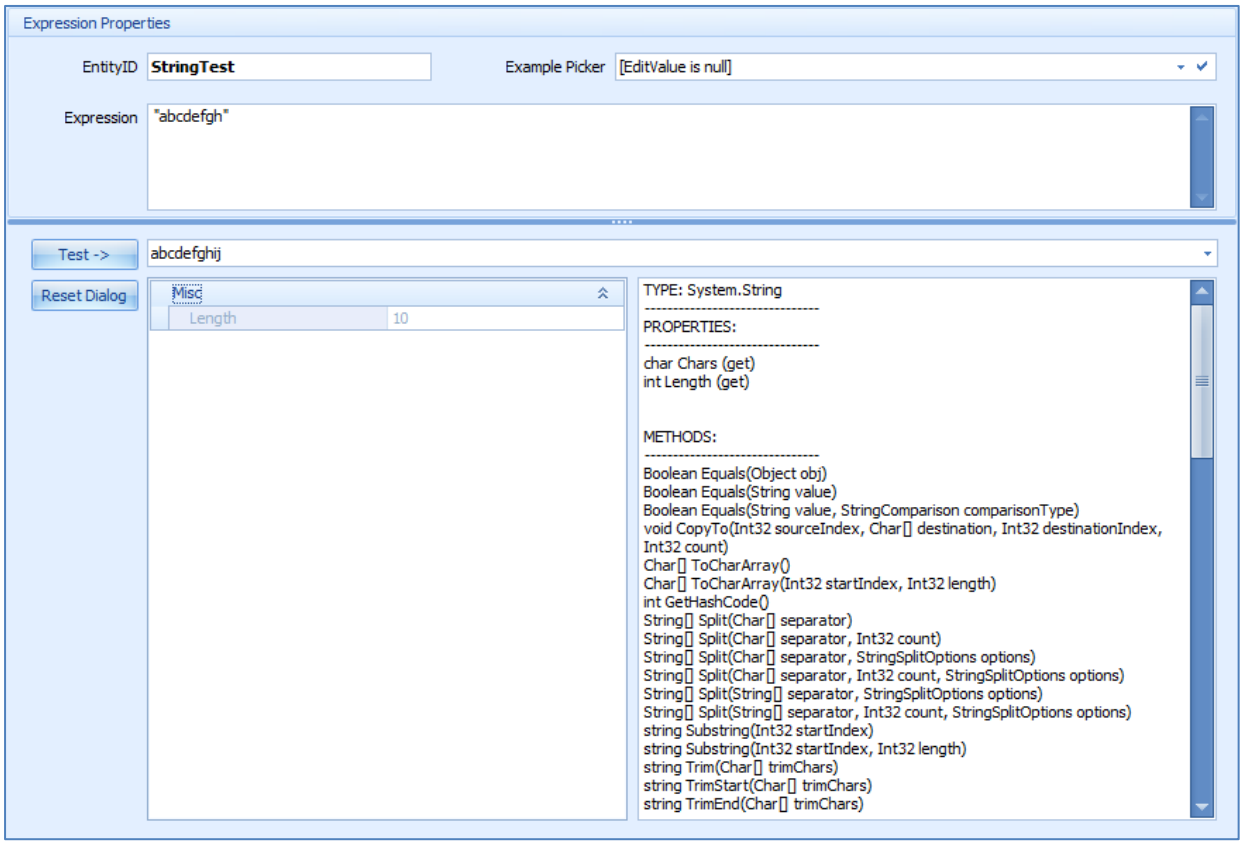
The end result will look something like this in Proteus:



Chapter 9: Defining Expression

Using Expressions when defining Workspaces in Proteus greatly increase the options for creating "dynamic" functionality, and used in conjunction with Custom Dialogues (see next chapter), it open up a wide range of possibilities.

It is important to know that Expressions (as well as Dialogs, DataSources and DataConnections) are available across *ALL* Workspaces, so they can in some cases be reused.



The screenshot above shows a very simple expression, and we shall use it as a starting point for getting a feel for this quite extensive functionality inherent in Expressions:
First of all, notice that we enclosed the text "abcdefgh" in double-quotes, as opposed to using single quotes in a dynamic expression for a calculated column.
When we press "Test ->", we see, as expected, the result "abcdefgh" (without quotes).
The first thing to point out here is the large amount of text that appears in the lower right textbox. It actually lists all the Properties and ***Methods*** for the resulting .NET object type, in this case the "System.String" (Note that it says "**TYPE: System.String**" at the top).
We will notice at the bottom of the screenshot a **method** that is quite useful called "**Substring**", and we see that there are different versions having different number of parameters. Let's pick this one:
Substring(startindex, length)
A method in .NET is applied to an object and the syntax is always:
Object.**Method**

We can then try this out for ourselves, by adding it to our existing expression – which is a string object:
"abcdefgh".**Substring(3,2)**

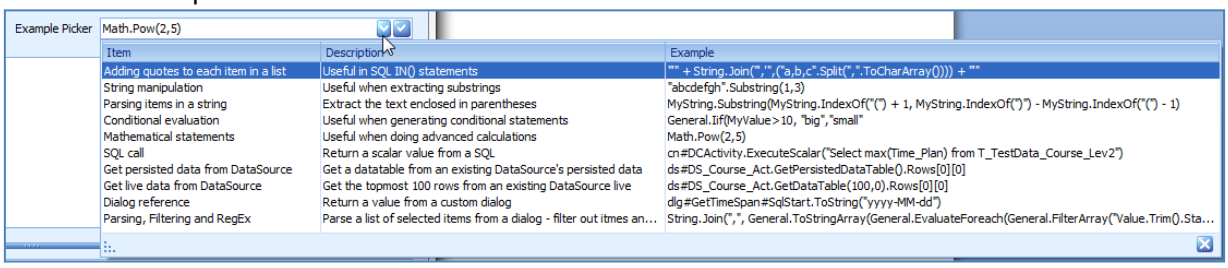
The result will be (notice that 3 means "the 4th position", since .NET consistently uses zero based indexing – i.e 0 as first position):

Some more Examples:

Purpose	Expression
Date functions:	
Get difference in days between 2 dates	DateValue1.Subtract(DateValue2).TotalDays
Add 7 days to a date	DateValue1.AddDays(7)
An explicit date	[2011-02-25]
Today's date	DateTime.Today
Parse a text to a date using the current culture	DateTime.Parse("30.1.13") -> Result is: **30.01.2013 00:00:00**
Formatting a date	DateTime.Today.ToString("yyyy-MM-dd")
Conditional testing:	
Test if one number is greater than another	General.Iif(MyValue>10, "Large","Small") + " Value."
Create a dynamic Iif statement based on database content	General.GetDynamicIifExpressionFromDataTable(ds#DS_ColorMap_Noise.GetDataTable(true), "0")
Database call:	
Lookup the highest value in a table	cn#MyConnection.ExecuteScalar("Select max(Mhrs) from MyTable")
Get a table of values from database, and picking the topmost row's value from column "Planned".	ds#MyDataSource.GetDataTable(true).Rows[0]["Planned"]
String functions:	
Replace all "A"s with "B"s in a string	"abcdefgabc".Replace("a","b")
Add quotes to all items in a list	""" + String.Join("',"',("a,b,c".Split(",".ToCharArray()))) + """
Alternative method for above	""" + "a,b,c".Replace(",","', "') + """
Inline arrays:	
Select the 3rd item in an array of constants	{101,102,103,104,105}[2]
Select the Nth item in an array of variables based on another variable	{R1_Field,R2_Field,R3_Field,R4_Field,R5_Field,R6_Field,R7_Field}[wpn]
Info on Workspace:	
Find who created the loaded Workspace	Repository.LoadedWorkspaces[0].CreatedBy
Find if current user is member of a set of Groups	Repository.AccessControl.IsMemberOf({"Admins", "MyGroup"})
Info on Environment:	
Get current windows login	Environment.UserName

Some help with built-in examples from Example Picker

There are some more examples to look at in the Example Picker that may give an indication for how to do various expressions:


Example Picker
Math.Pow(2,5)
Item
Description
Example
Adding quotes to each item in a list
Useful in SQL IN() statements
"" + String.Join("","",("a,b,c".Split(",".ToCharArray()))) + ""
String manipulation
Useful when extracting substrings
"abcdefgh".Substring(1,3)
Parsing items in a string
Extract the text enclosed in parentheses
MyString.Substring(MyString.IndexOf("(") + 1, MyString.IndexOf(")") - MyString.IndexOf("(") - 1)
Conditional evaluation
Useful when generating conditional statements
General.Iif(MyValue>10, "big","small"
Mathematical statements
Useful when doing advanced calculations
Math.Pow(2,5)
SQL call
Return a scalar value from a SQL
cn#DCActivity.ExecuteScalar("Select max(Time_Plan) from T_TestData_Course_Lev2")
Get persisted data from DataSource
Get a datatable from an existing DataSource's persisted data
ds#DS_Course_Act.GetPersistedDataTable().Rows[0][0]
Get live data from DataSource
Get the topmost 100 rows from an existing DataSource live
ds#DS_Course_Act.GetDataTable(100,0).Rows[0][0]
Dialog reference
Return a value from a custom dialog
dlg#GetTimeSpan#SqlStart.ToString("yyyy-MM-dd")
Parsing, Filtering and RegEx
Parse a list of selected items from a dialog - filter out itmes an...
String.Join(",", General.ToStringArray(General.EvaluateForeach(General.FilterArray("Value.Trim().Sta...

If you are familiar with the .NET Class Library, you will recognize the ones that are available in the Expression builder:
String, DateTime, Math, Regex, Environment, File, Directory, Path, Process, Convert

There are a few Class Libraries that are developed specifically for Proteus, and they are accessible through these names:
General, Calendar, TextConverter, Repository, Registry, Color, DataLayout, Outlook

To find out more on each of these, simply type the name of the Class Library as the Expression and press “Test ->”. You will then, as we saw earlier, get a list of all the methods that are available for it.

Some examples of non-trivial Expressions

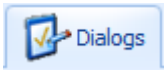
The scope of this document is certainly not wide enough to give a full insight into all the objects and their methods that are available to the person building Workspaces, and must be left for special advanced courses (can be requested at proteus@promineo.no, or alternatively from your regional Proteus retailer).

Just to give a flavor for the possibilities here, a few "non-trivial" expressions are supplied here (from actual Workspaces built by author):

Purpose	Expression
Set the values of 4 specific columns, for each row of a DataSet, with the periodized result of some of its other column values	`MainDataSet.Tables["Resource"].SetColumnValuesFromExpression({"target_cost_early", "target_cost_late", "target_qty_early", "target_qty_late"}, {"MainDataSet.Periodizer.GetScopeLeftOfCutoff(General.ToDouble(target_cost), General.IsNull(_Parent_clndr_id, -1), General.IsNull(curv_id, -1), SnapshotDate, target_start_date, target_end_date)", "MainDataSet.Periodizer.GetScopeLeftOfCutoff(General.ToDouble(target_cost), General.IsNull(_Parent_clndr_id, -1), General.IsNull(curv_id, -1), SnapshotDate, rem_late_start_date, rem_late_end_date)", "MainDataSet.Periodizer.GetScopeLeftOfCutoff(General.ToDouble(target_qty), General.IsNull(_Parent_clndr_id, -1), General.IsNull(curv_id, -1), SnapshotDate, target_start_date, target_end_date)", "MainDataSet.Periodizer.GetScopeLeftOfCutoff(General.ToDouble(target_qty), General.IsNull(_Parent_clndr_id, -1), General.IsNull(curv_id, -1), SnapshotDate, rem_late_start_date, rem_late_end_date)" })`
Find max value of a column value given a certain filter condition	`MainDataSet.Tables["Noise"].LookupValue("RevisionNo DESC","WEACRef = '{PrimKey}'") + 1`
Launch a custom export	`Repository.LoadedWorkspaces[0].DataSets[0].ExcelExportItems[7].CreateReport(WEACTree.GridView)`
Get the date of the most recent Sunday.	`Calendar.GetFirstCurrentDate("w", DateTime.Today).AddDays(-1)`
Create a filename on the temp folder,	`Environment.GetEnvironmentVariable("TMP") + "\\TempFile.txt"`

Chapter 10: Defining custom Dialogs

Custom Dialogs, in conjunction with Expressions, add an endless range of possibilities to make interactions with users more dynamic and flexible.

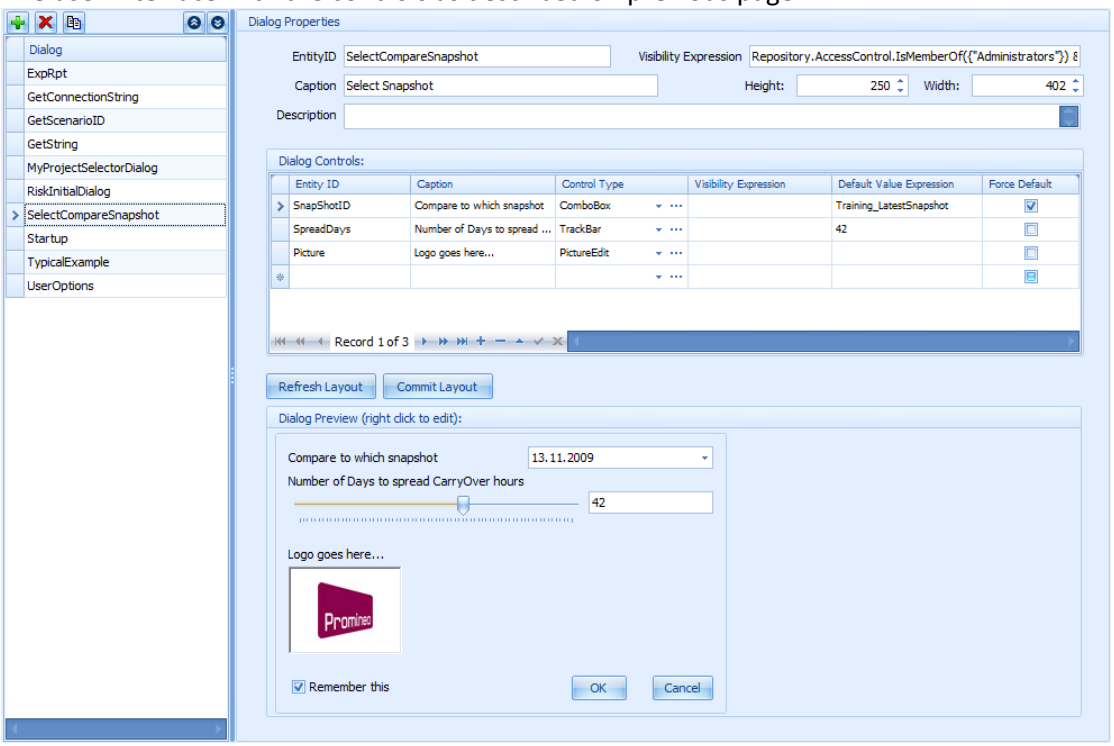


A Custom Dialog is functionally very similar to a DataLayout control in Proteus, only it does not support all the controls (such as Gauges and Graph), and is shown in a popup dialog form instead of in a dockable tab.

Let's first list all the controls in user interface, and then take the settings in the screenshot below, as an example to get started:

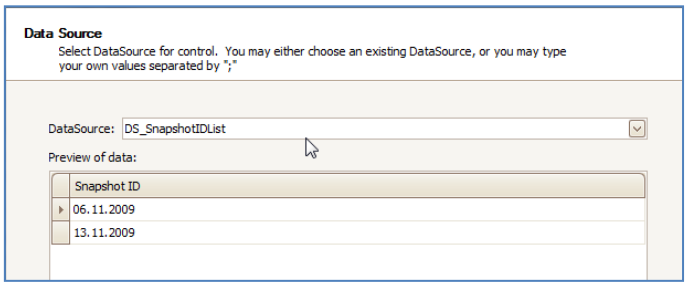
Control	Description
EntityID	Give the Dialog a unique ID.
Caption	The text showing as caption in popup dialog window.
Description	For internal documentation.
Visibility Expression	An expression that can control who should see this dialog. If it evaluates to "false", it will not be seen by the user, and default settings (see below for more info) will be used.
Height / Width	Control the dimensions of the dialog.
Refresh Layout button	This is pressed after Control properties have been modified, or the Control Type has been changed, so that is reflects in Preview.
Commit Layout button	This button must be pressed after layout has been modified in order for changes to be persisted (remembered).
Dialog Controls grid	See column descriptions below.
Columns in the Dialog Controls grid:	
Entity ID	The EntityID for the control.
Caption	The caption text to show for control.
Control Type	Which type of control to use: Similar to list for DataLayout.
Visibility Expression	An expression that controls the visibility of control. If you want the dialog to be visible to a select Group, you could (as in the screenshot) supply this expression: Repository.AccessControl.IsMemberOf({"DlgDemoMember"})
Default Value Expression	An expression that controls the initial value of control. If you wish to suggest the most recent snapshot, you may create an Expression that makes a call to the database to get the ID for the most recent snapshot, and then here refer to that Expression's Name. In the example below we have the Expression named "Training_LatestSnapshot", with this definition: `cn#DCSnapshot.ExecuteScalar("Select Max(SnapshotID) From Snap_ProteusTraining_Activities")`
Force Default	If checked, the above expression will always be set as control's initial value. If unchecked, the initial value is remembered from last time (if "Remember this" is checked at lower left corner of dialog). If this had been checked (which it is) for the example in row above, the user would always have been suggested the most recent snapshot, and not the last remembered value selected by user.

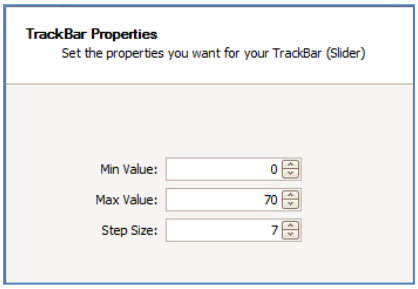
The user interface with the controls as described on previous page:



Screenshot of the form for defining Custom Dialogs.

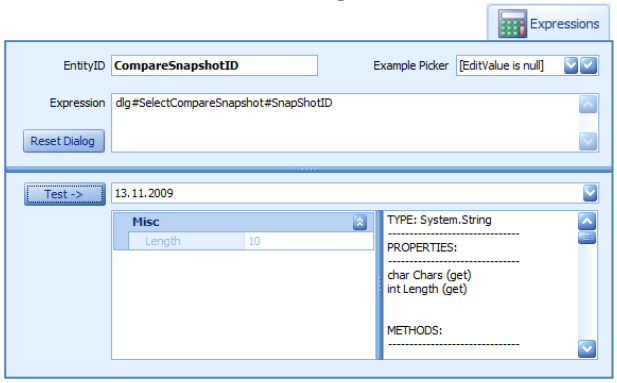
When clicking the ellipses (button with dots) for the ControlType "ComboBox", the wizard guides you through some choices (such as which DataSource to get data from):


Data Source
Select DataSource for control. You may either choose an existing DataSource, or you may type your own values separated by ";"
DataSource: DS_SnapshotIDList
Preview of data:
Snapshot ID
06.11.2009
13.11.2009



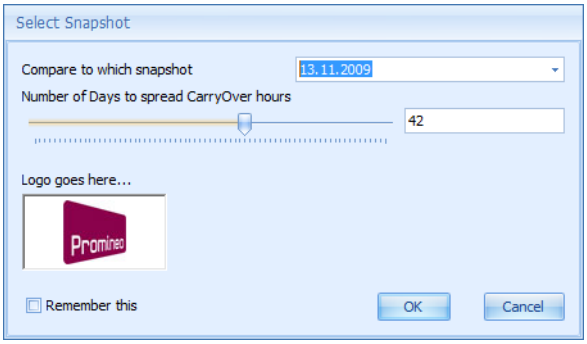
Similarly, the wizard will help you define the properties for the TrackBar control:

How to test a Custom Dialog?



The Expression that refers to dialog control will bring up the dialog form, and return the requested value given by user.

It is important to point out that the values supplied to a Custom Dialog are ***cached***, so that next time the dialog value is requested, it *returns the cached value*.
In order to ensure that the dialog will be shown, the "**Reset Dialog**" must be pressed before the "**Test ->**" button is pressed.



Chapter 11: Setting up Action Buttons

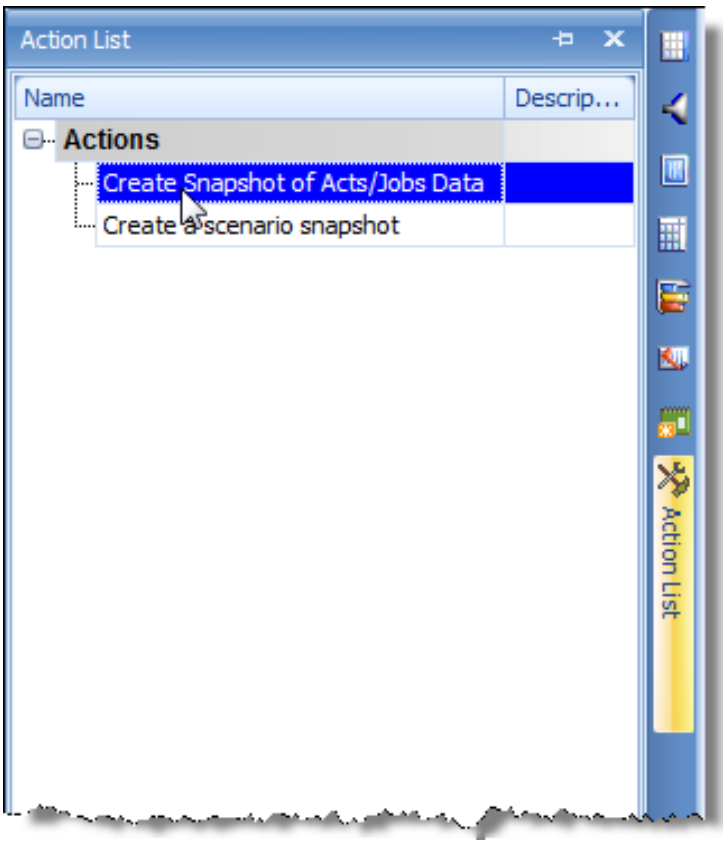
The idea behind "Action Buttons" in Proteus is to give Administrators the possibility to "wire up" needed functionality into one or more "buttons" that certain Groups could have access to. The current possibilities for these are listed below and each is described in detail. This list will most likely be extended in later versions of Proteus.

Action Item Type	Description
Publish	Publish data to external DataConnection using SQL Statement(s)
PublishSnapshot	Publish data to external DataConnection using current contents (or limited to a set of fields) of specific (or all) loaded DataSet(s).
PublishGrid	Publish data to external DataConnection using content of a particular DataSet's filtered rows in grid. For each row an update statement may be built to append data to some table.
PublishImport	Publish data to external DataConnection using Excel Workbook as source for SQL statements (similar to above but uses Excel content instead of filtered rows in grid).
Expression Action	Evaluates an Expression (that carries out an action).

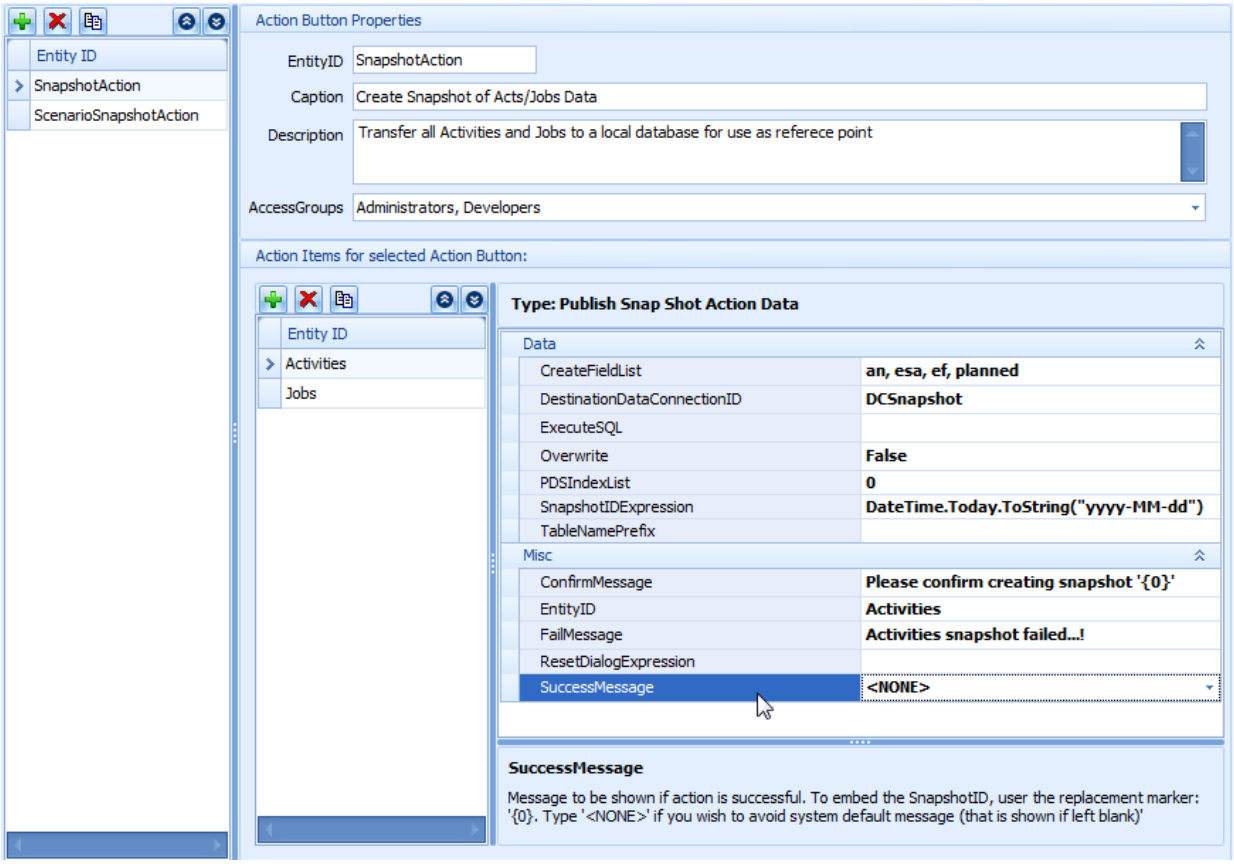
What is an "Action Button"?

An Action Button represents an entry in the Proteus "Action List" Tree (normally configured to be accessible on right hand side of screen – see screenshot).

It can consist of one or more so called "Action Items", where each Action Item is one of the types described in the table above.



Below is a screenshot of the form for defining Action Buttons. Each selected row in the property grid will give show a descriptive message of what it does at bottom of screen.

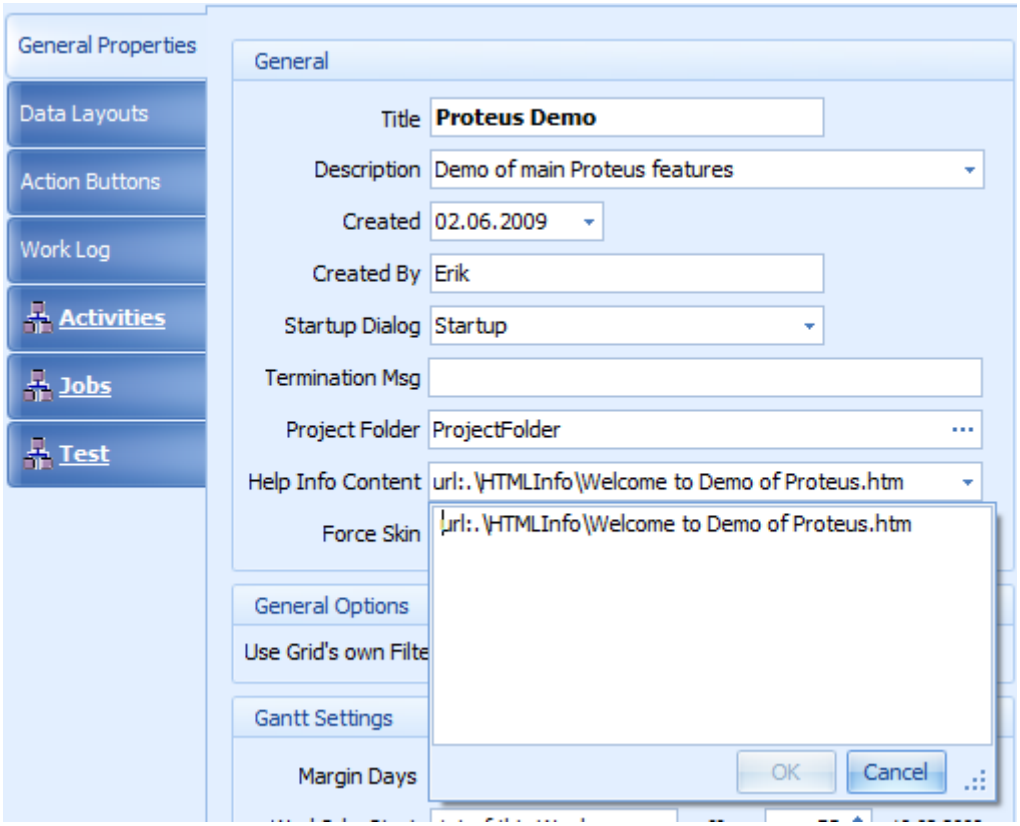


Screen shot showing Action Button ("SnapshotAction") with 2 Actions-items ("Activities" and "Jobs")

Property	Description
EntityID	Give the Action Button a unique ID.
Caption	The text to show in the Action List.
Description	The text shown for the tooltip in the Action List.
Access Groups	Control which Groups will have access to this Action Button
Action Items for selected Action Button:	
The set of properties depends on which type of Action Item type is created, but using the information given (at bottom) after selecting an item in the property grid should make things fairly clear.	

Chapter 12: Creating Help information for a workspace

One Workspace in Proteus can be quite different from another, and can be considered an "application" in its own right. It is therefore useful to be able to provide "built in documentation" for each Workspace.

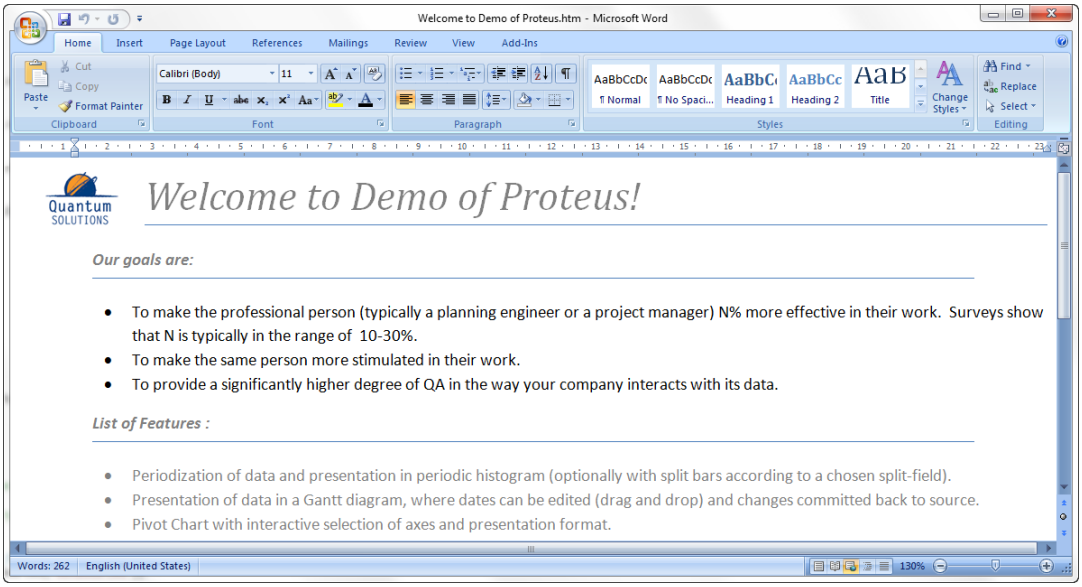


The General tab of a Workspace form allows entering a reference to its help content.

There are three options when defining the HTML content:

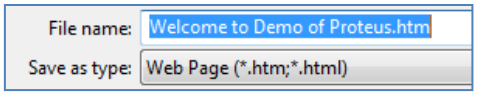
1. A reference to a html file, using the "url:" prefix (recommended)
2. A reference to a word file, using the "doc:" prefix.
3. Entering HTML code directly in the textbox (not recommended)

In the example above a repository subfolder has been created called "HTMLInfo" and there the .htm-file (and related folder) is located.


Welcome to Demo of Proteus.htm - Microsoft Word
Home
Insert
Page Layout
References
Mailings
Review
View
Add-Ins
Quantum SOLUTIONS
Welcome to Demo of Proteus!
Our goals are:
To make the professional person (typically a planning engineer or a project manager) N% more effective in their work. Surveys show that N is typically in the range of 10-30%.
To make the same person more stimulated in their work.
To provide a significantly higher degree of QA in the way your company interacts with its data.
List of Features :
Periodization of data and presentation in periodic histogram (optionally with split bars according to a chosen split-field).
Presentation of data in a Gantt diagram, where dates can be edited (drag and drop) and changes committed back to source.
Pivot Chart with interactive selection of axes and presentation format.
Words: 262
English (United States)
130%

Functional technique: Write the content in Microsoft Word and save it as html.

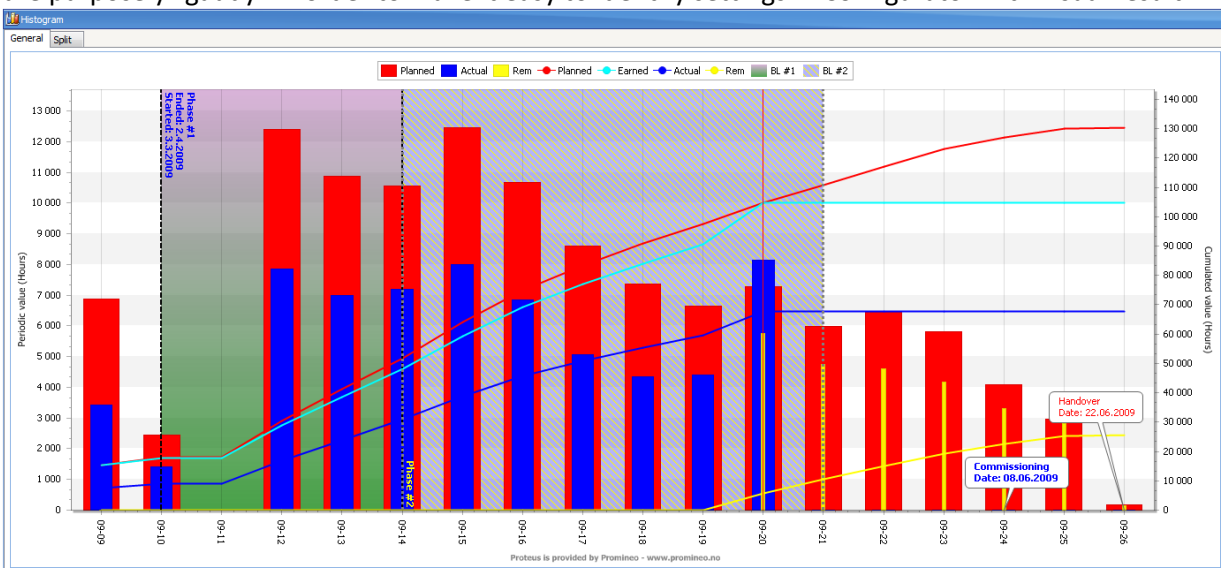
Microsoft Word is a good tool to create the content and then save it as web page:



Chapter 13: Adding visual information using Annotations

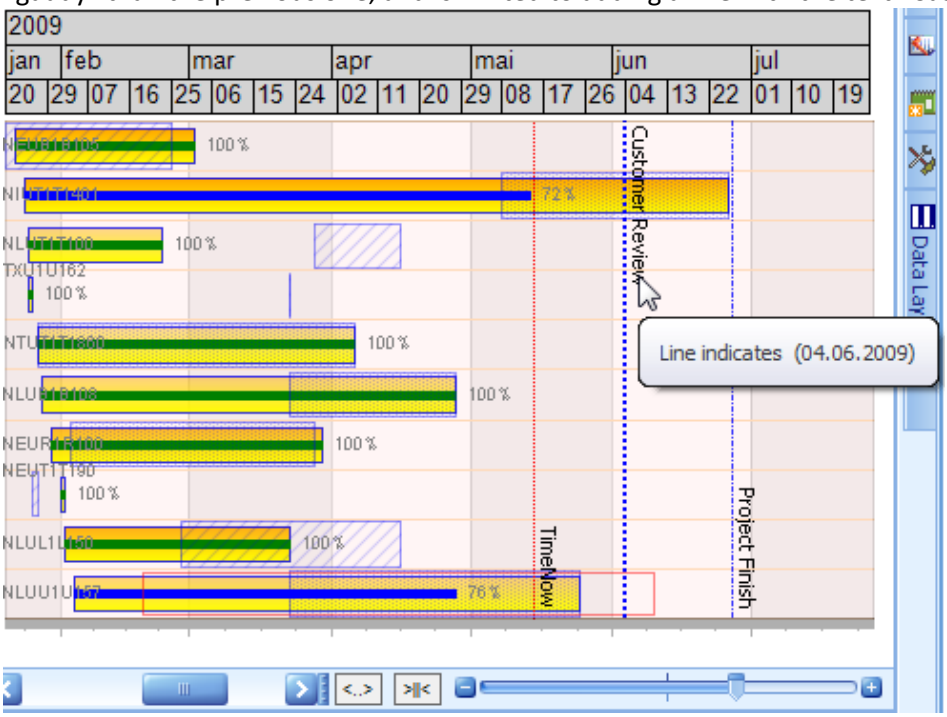
It may sometimes be useful to highlight certain points in time, or intervals of time in a Histogram, Gantt Chart or Graph (in a DataLayout). In this chapter we will look into how this is achieved and how custom lines, stripes, and callouts can be defined.

Let's look at how we can achieve the annotations shown below in a histogram. The visual elements are purposely "gaudy" in order to make it easy to identify settings in Configurator with visual result:



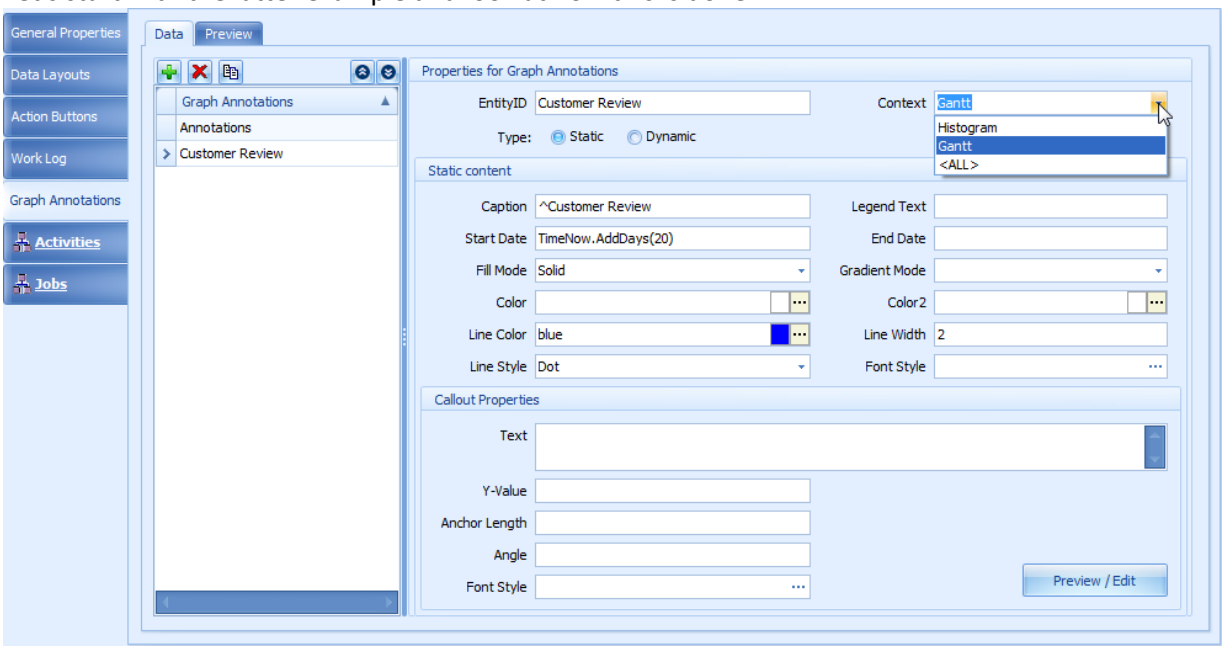
Four annotations are shown in Histogram: Two vertical colored strips and two callouts with text.

Let's also look at how we can make similar changes to a Gantt Chart. This example is much less "gaudy" than the previous one, and is limited to adding a line with the text "Customer Review":



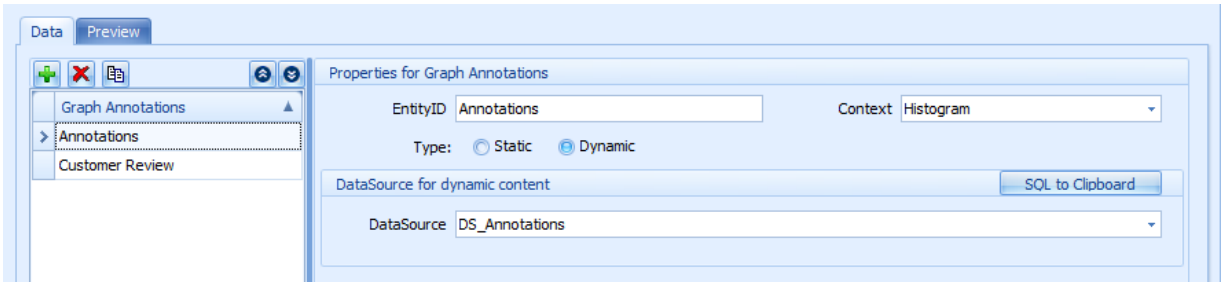
One annotation has been added to Gantt Chart: A vertical line saying "Customer Review".

Let's start with the latter example and look at how this is done:



As you can see, there are quite a few settings that can be made for an annotation, and you can get information about each by watching the tooltips shown when hovering over their labels.
We will instead focus here on the settings required to achieve the blue dotted line with the text "Customer Review" and at its particular location in time:
First, notice that the "Context" is set to "Gantt". This will ensure that the selected annotation (in this case the "Customer Review" line) will only appear in the Gantt Chart.
In the Caption box we have typed "^Customer Review", and the "^" means that it is top aligned.
Please note that all settings can reference an Expression using the curly bracket syntax (you could e.g have typed "^Customer Review: {AdditionalInfoExpression}").
The "Start Date" holds the expression that results in the date to use for the line (and correspondingly, the "End Date" for a strip): "TimeNow.AddDays(20)"
This results is 20 days ahead of whatever the Expression "TimeNow" evaluates to.

Getting back to the Histogram annotations: They have been defined in a database, so the Configurator only has one entry for all of these, and it has its "Type" set to "Dynamic" – indicating that it is defined using a DataSource (in this case "DS_Annotations"). The format for such a source is described in Appendix. It is worth mentioning here that the button "SQL to Clipboard" (see screenshot below) will place a starting point for making the SQL statement, showing all the required fieldnames.



The Dynamic Annotation gets its data from a DataSource. "SQL to Clipboard" will give a starting point for defining SQL.

The next page shows the 4 rows of data used to produce the annotations in the histogram (2 stripes and 2 callouts), and all its column values (shown over 2 rows for readability).

Caption	LegendText	FontStyle	AnnotationText	AnnotationYValue	AnnotationLength	AnnotationAngle
^Phase #1<CR>Ended: 2.4.2009<CR>Started: 3.3.2009	BL #1	Bold§Blue				
Phase #2	BL #2	Bold§Yellow				
			Commissioning<CR>Date: 08.06.2009			
			Handover<CR>Date: 22.06.2009		120	105

AnnotationFontStyle	StartDate	EndDate	FillMode	GradientMode	Color	Color2	LineColor	LineWidth	LineStyle	Context
	03.03.2009	02.04.2009	Gradient	BottomToTop	green,70%	purple,30%	black	2	Dash	<ALL>
	02.04.2009	20.05.2009	Hatch	ForwardDiagonal	blue,30%	yellow	gray	3	Dot	<ALL>
Bold§Blue	08.06.2009									
Regular§Red	22.06.2009									

The 4 rows of data coming from DS_Annotations DataSource that results in 2 colored strips, and 2 callouts with text.

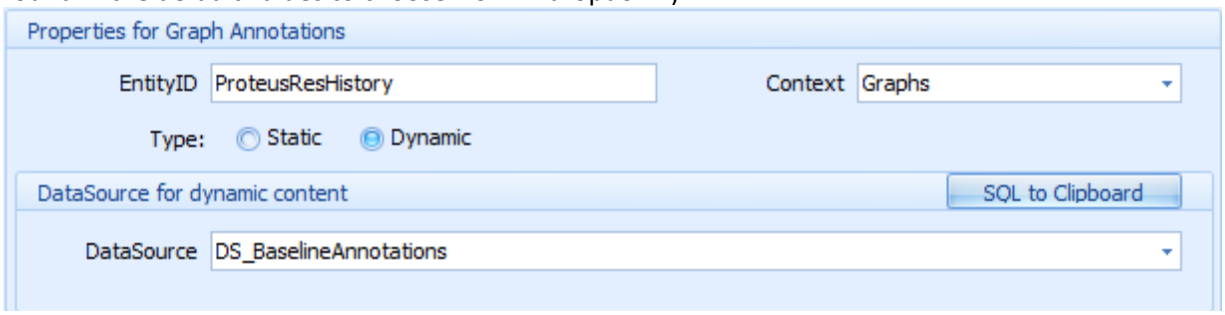
Notes:

In the "Caption" column you see the text "**<CR>**" which results in a line feed (Carriage Return) in the text. The same is true for "AnnotationText".

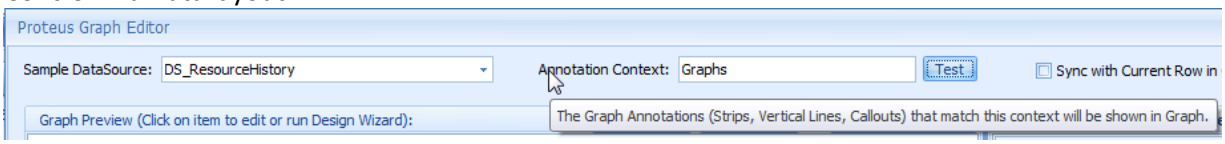
Note that the "**Context**" column, if provided, can be used to control where the annotation should be shown. If the field is omitted from DataSource, the Context dropdown is used to control where it appears, and if left blank, it will have the same effect as the "**<ALL>**" setting

How to get annotations in a DataLayout Graph

Let's say we have an annotation item which has a "Context" setting of e.g "Graphs" (a typed value not found in the default values to choose from in dropdown):

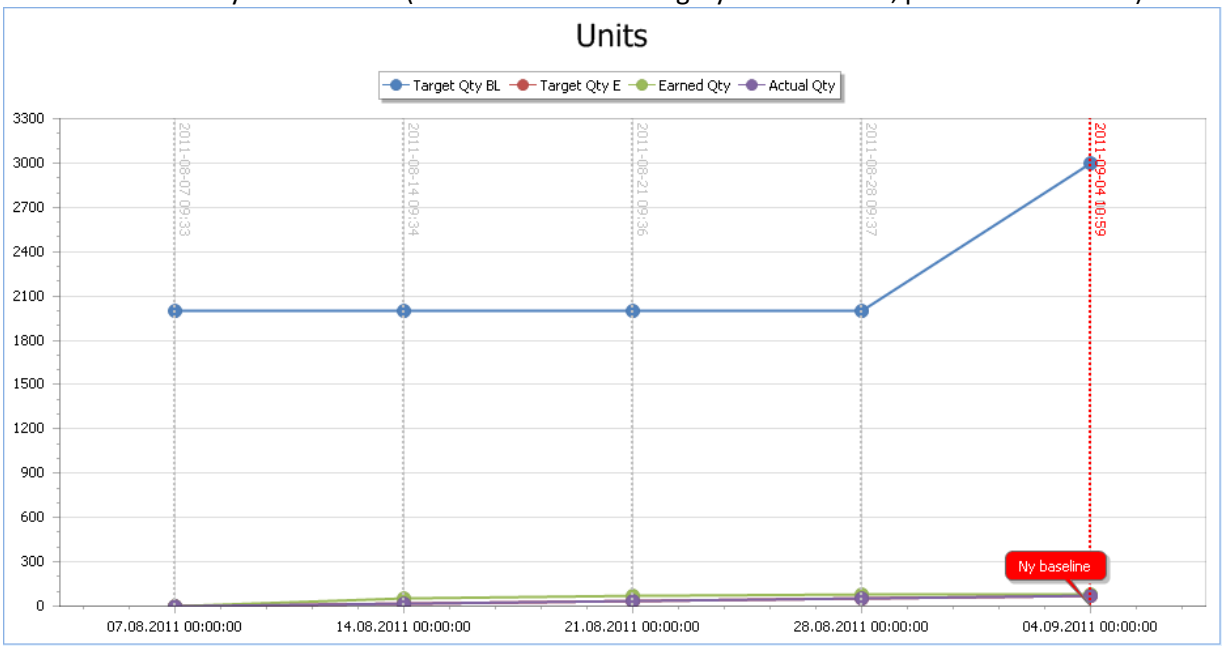


You can then match this value in the "**Annotation Context**" textbox found in the editor for the Graph Control in a DataLayout:


Proteus Graph Editor
Sample DataSource:
DS_ResourceHistory
Annotation Context:
Graphs
Test
Sync with Current Row in
Graph Preview (Click on item to edit or run Design Wizard):
The Graph Annotations (Strips, Vertical Lines, Callouts) that match this context will be shown in Graph.

..

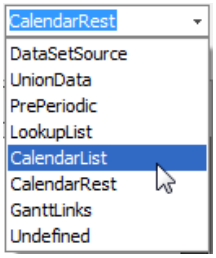
Then the result may look like this (the vertical lines with gray and red texts, plus the red callout):



Result showing 5 vertical lines (one is red, and the others gray), and a red callout with white text.

Chapter 14: Appendix A – DataSource formats for Calendars

If you need to specify a set of calendars to be used when time-phasing data in Proteus, this appendix describes how to accomplish this. Proteus currently supports calendar formats for Safran Project, Safran Planner and Primavera (P6).



Setting up calendars for Safran Project:

You need to set up two separate DataSources, where one is used to specify calendar names and workhours per day, and the other to specify non-working days. It is important to set the "**Used in Context**" type for each of these to "**CalendarList**" and "**CalendarRest**" respectively.

Some sample data:

Calendar List (defining list of calendars and their working hours per day):

wpn	description	hours
	Click here to add a new row	
0	For System Use Only	7,5
1	Engineering	7,5
2	Fabrication - Stavanger	6,5
3	Fabrication - Egersund	7,5
4	Offshore 1	10,5
5	Offshore	11

Calendar Rest (defining non-working days for each calendar):

wpn	rest_from	rest_until	day_of_week
2	01.01.2009	01.01.2011	6
2	01.01.2009	01.01.2011	7
2	24.12.2010	02.01.2011	8

Some details:

The fieldnames for each of the DataSources are as shown above, and their field-types are as follows:

wpn: Integer
description: Text
hours: float
rest_from, rest_until: DateTime
day_of_week: Integer

The column "day_of_week" controls which weekday to "block from work" in the range specified by "rest_from" and "rest_until". The number **1** represents **Monday**, **2 Tuesday**, etc, and **8** implies "**All Days in week.**"

Setting up calendars for Primavera (P6):

You need to set up one DataSources (and not two as for Safran), having its "**Used in Context**" type set to "**CalendarList**".

Below is an example using SQL (Note that "{Selected_Projects}" refers to result of an Expression):

```
SELECT
 c.clndr_id,
 clndr_name,
 clndr_data,
 day_hr_cnt
FROM
 CALENDAR c
Where
 c.clndr_id in (Select distinct t.clndr_id from TASK t inner join
PROJECT p ON t.proj_id = p.proj_id WHERE p.proj_id
IN({Selected_Projects}))
```

Calendar List:

clndr_id	clndr_name	clndr_data	day_hr_cnt
597	Standard 5 ...	(0||CalendarData()((0||Da...	8,00

Note:

Some versions of Primavera do not have the "**day_hr_cnt**" field, and then this can be omitted - Proteus will deduce the number of workhours per day from the content of the "**clndr_data**" column, containing the encoded intervals for each day of week, as well as a list of dates and time-intervals for non-work (restrictions).

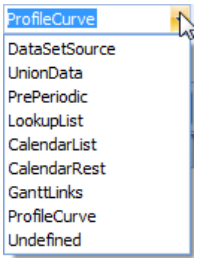
Chapter 15: Appendix A2 – DataSource formats for Profiles

If you need to specify a set of curve profiles to be used when time-phasing data in Proteus, this appendix describes how to accomplish this. Proteus currently supports profile formats for Safran Project and Primavera (P6).

Setting up profiles for Safran Project:

You need to set up a DataSource to specify profile definitions. It is important to set the "Used in Context" to "**ProfileCurve**".

Here is an example:



Profile definitions (with "coordinates" for producing the shape of the profile):

profile_id	value_x	value_y
1	10	13
1	50	73
1	75	80
1	100	100
2	10	13
2	50	73
2	75	80
2	100	100
3	40	40
3	76	90
3	100	100

Data Types:

profile_id: Integer
value_x: Double
value_y: Double

Setting up profiles for Primavera (P6):

You need to set up a DataSource with its "Used in Context" type set to "**ProfileCurve**".

Below is an example of SQL:

```
SELECT
  curv_id,
  curv_data
FROM
  RSRCCURV
```

Calendar List:

T_Primavera_Profile

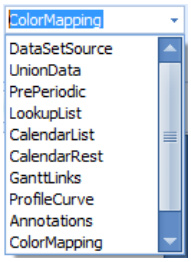
curv_id	curv_data
1	(0||CurveData()((0||0(PctUsage|0)()) (0||1(PctUsage|5)()) (0||2(PctUsage|5)()) (0||3(PctUsage|5)()) (0||4(PctUsage|5)()) (0||5(PctUsage|5)()) (0||6(PctUsage|5)()) (0||7(PctUsage|5)()) (0||8(PctUsage|5)()) (0||9(PctUsage|5)()) (0||10(PctUsage|5)()) (0||11(PctUsage|5)()) (0||12(PctUsage|5)()) (0||13(PctUsage|5)()) (0||14(PctUsage|5)()) (0||15(PctUsage|5)()) (0||16(PctUsage|5)()) (0||17(PctUsage|5)()) (0||18(PctUsage|5)()) (0||19(PctUsage|5)()) (0||20(PctUsage|5)()))
2	(0||CurveData()((0||0(PctUsage|0)()) (0||1(PctUsage|3.5)()) (0||2(PctUsage|3.5)()) (0||3(PctUsage|3.5)()) (0||4(PctUsage|3.5)()) (0||5(PctUsage|3.5)()) (0||6(PctUsage|3.5)()) (0||7(PctUsage|3.5)()) (0||8(PctUsage|3.5)()) (0||9(PctUsage|3.5)()) (0||10(PctUsage|3.5)() (0||11(PctUsage|6.5)()) (0||12(PctUsage|6.5)()) (0||13(PctUsage|6.5)()) (0||14(PctUsage|6.5)()) (0||15(PctUsage|6.5)()) (0||16(PctUsage|6.5)()) (0||17(PctUsage|6.5)()) (0||18(PctUsage|6.5)()) (0||19(PctUsage|6.5)()) (0||20(PctUsage|6.5)()))
3	(0||CurveData()((0||0(PctUsage|0)()) (0||1(PctUsage|0.5)()) (0||2(PctUsage|0.5)()) (0||3(PctUsage|1.5)()) (0||4(PctUsage|1.5)()) (0||5(PctUsage|4)()) (0||6(PctUsage|4)()) (0||7(PctUsage|7.5)()) (0||8(PctUsage|7.5)()) (0||9(PctUsage|11.5)()) (0||10(PctUsage|11.5 (0||11(PctUsage|11.5)()) (0||12(PctUsage|11.5)()) (0||13(PctUsage|7.5)()) (0||14(PctUsage|7.5)()) (0||15(PctUsage|4)()) (0||16(PctUsage|4)()) (0||17(PctUsage|1.5)() (0||18(PctUsage|1.5)()) (0||19(PctUsage|0.5)()) (0||20(PctUsage|0.5)()))

Chapter 16: Appendix A3 – DataSource for Custom ColorMapping

It is possible to map a particular value of a field to a particular color when viewing split histograms. In this way you may, for example, map a limited set of codes (e.g phase codes) to a corresponding set of colors, in order to avoid getting random coloring schemes each time a histogram is created.

Setting up Custom SplitBar Colors:

You need to set up a DataSource to specify such color mapping definitions. It is important to set the "Used in Context" to "**ColorMapping**".
Here is an example:



Split Field Name	Split Value	Split Color
R2_Field	L	Red
R2_Field	N	Green
R2_Field	T	Blue
R2_Field		Yellow
R1_Field	XX	Red

The field names are arbitrary, but the ***order*** of the fields is critical (from left to right):

1. Name of the split-field
2. The split value
3. The split color (see naming conventions for coloring)

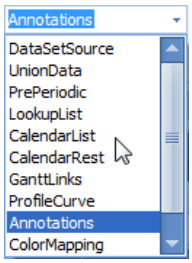
Note:

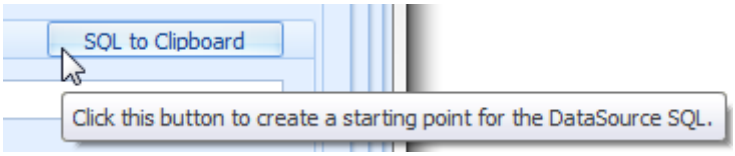
The name of the split field (see 1. above)follows that if you have the same fieldname in two DataSets it will use the same color map. This will in most cases be an advantage, but if you want to avoid such situations, you need to create unique names for the fields across your DataSets.

Chapter 17: Appendix A4 – DataSource for Annotations

As shown previously in this document, Annotations can be used to inform users of important information in Histogram, Gantt Chart, or a Graph in a DataLayout. If you have chosen a "Dynamic" type of Annotation, then you need to supply a DataSource that gives the final definition of how they should be rendered. This Appendix explains the format of such a DataSource.

Notice that the DataSource must be categorized as "Annotations" in order to be selectable as an Annotation DataSource.





A quick way to get started is to click the button shown above, and then paste the result as the starting point for the SQL in the DataSource editor:

```
Select
'Last Cutoff' as Caption,
'' as LegendText,
'§red' as FontStyle,
'' as AnnotationText,
'' as AnnotationYValue,
'' as AnnotationLength,
'' as AnnotationAngle,
'' as AnnotationFontStyle,
'{TimeNow}' as StartDate,
null as EndDate,
'' as FillMode,
'' as GradientMode,
'' as Color,
'' as Color2,
'Red' as LineColor,
'2' as LineWidth,
'Dot' as LineStyle,
'<ALL>' as Context
FROM
 MyTable
```

Note that you may *omit the columns that are not required for your purposes*. If the Context column is provided, you may control in which context each item is shown.
As seen from the SQL above, all the fields have the data type "String" (called different names for different databases types)
The content of the table may then look like this:

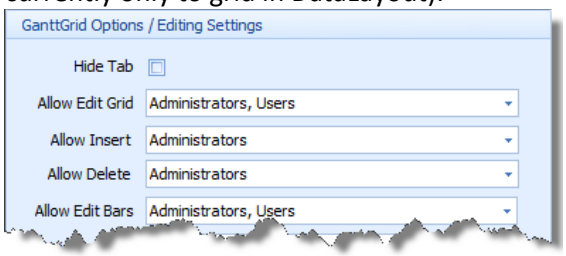
Caption	LegendText	FontStyle	AnnotationText	AnnotationYValue	AnnotationLength	AnnotationAngle
^Phase #1<CR>Ended: 2.4.2009<CR>Started: 3.3.2009	BL #1	Bold§Blue				
Phase #2	BL #2	Bold§Yellow				
			Commissioning<CR>Date: 08.06.2009			
			Handover<CR>Date: 22.06.2009		120	105

AnnotationFontStyle	StartDate	EndDate	FillMode	GradientMode	Color	Color2	LineColor	LineWidth	LineStyle	Context
	03.03.2009	02.04.2009	Gradient	BottomToTop	green,70%	purple,30%	black	2	Dash	<ALL>
	02.04.2009	20.05.2009	Hatch	ForwardDiagonal	blue,30%	yellow	gray	3	Dot	<ALL>
Bold§Blue	08.06.2009									
Regular§Red	22.06.2009									

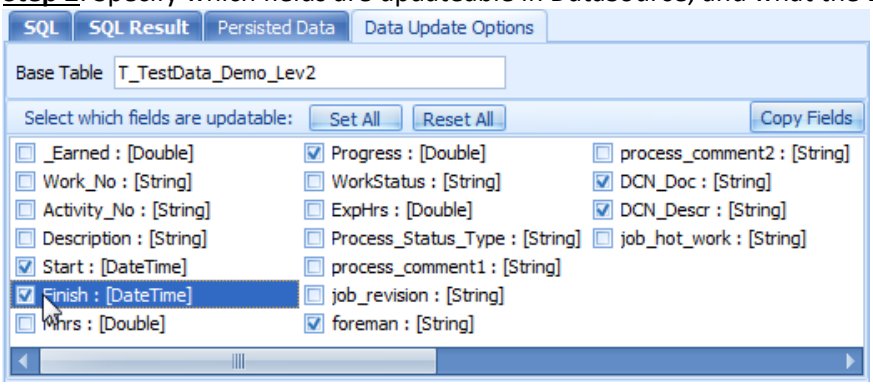
Chapter 18: Appendix B: How to modify source data from Proteus

Proteus not only allows reading source data, but also has the possibility to update it as well (providing that the data source allows updates). This appendix gives info on how this is set up.

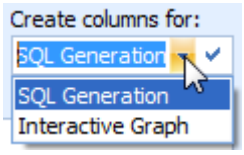
Step 1: In the DataSet's General Tab, set which Group(s) should have the right to update data in grid, move Gantt Bar using drag-and-drop, as well as insert and delete data (the latter two applies currently only to grid in DataLayout):

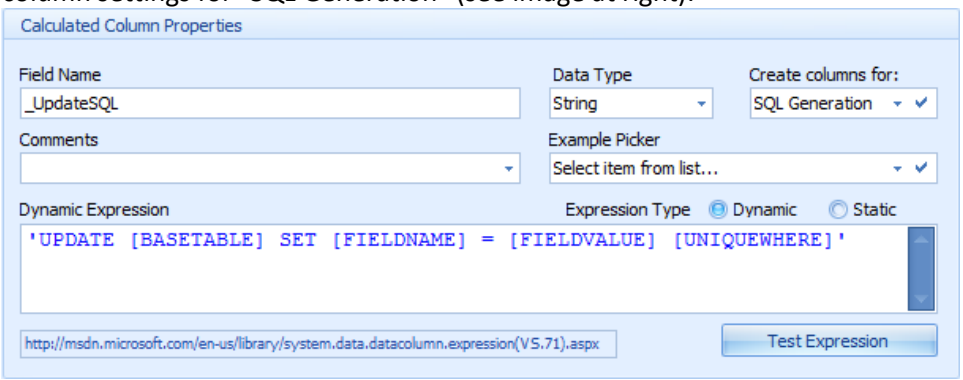


Step 2: Specify which fields are updateable in DataSource, and what the Base Table's name is:



Step 3: Specify how Update, Insert and Delete Statements are generated:
The simplest way to achieve this is to press the button that inserts standard column settings for "SQL Generation" (see image at right):

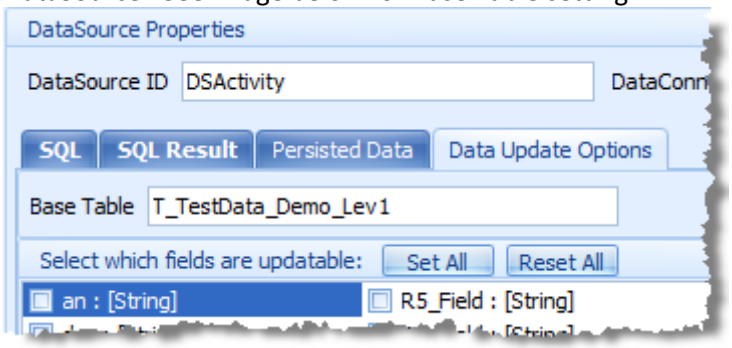




Typical (autogenerated) setting for calculated column "_UpdateSQL".

It will produce the 3 calculated columns, having the specialized names:
_UpdateSQL, **_InsertSQL**, and **_DeleteSQL**.
_UpdateSQL is a reserved fieldname to be used to define update SQL statement used when DataSet is edited:
It can have reserved markers: **[FIELDNAME]** and **[FIELDVALUE]**, **[UNIQUEID]**, **[UNIQUEWHERE]**, **[BASETABLE]**

Note 1: [FIELDVALUE] will be SQL formatted, so that if it is a string, it will automatically be surrounded by single quotes.
Note 2: The [BASETABLE] marker is replaced with the "Base Table" value set for the DataSet's DataSource. See image below for Base Table setting:



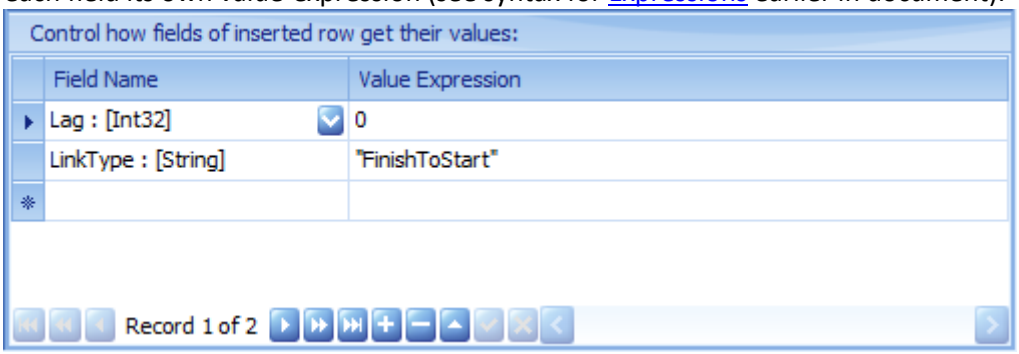
For each field marked as updatable (in Step 3), an Update SQL statement will be created, provided the field's value has been modified.

It is possible to create "Column Specific Update Statements" by naming the calculated column **_UpdateSQL_**<FieldName>, and this will have precedence over the generic **_UpdateSQL** definition.

Note: If you need a customized update SQL statement for updating the dates changed by dragging bar in Gantt Chart, then **_UpdateGanttSQL** can control this update. If not present, the **_UpdateSQL** will be used instead.

Step 4 (optional): Control how added rows get their initial value(s):

It is possible to control initial values in fields when new rows are added to a DataSet, by assigning each field its own value-expression (see syntax for Expressions earlier in document):



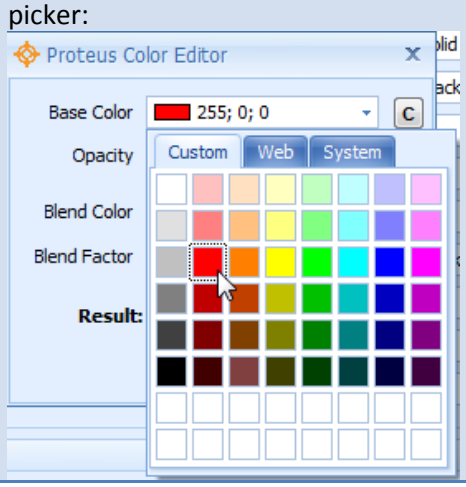
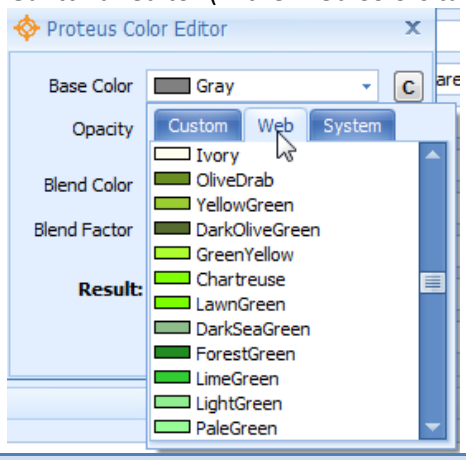
Advanced Note:
It is possible to have some special "Expressions" that results in the value from last added row or last selected row:
<LASTSELECTEDVALUE>
<LASTADDEDVALUE>

In some cases you need to generate the "next" number in a sequence based on your loaded data, i.e you must find the highest number and add one to it. This is possible with an expression like this:
MainDataSet.Tables["ADataSetNameHere"].LookupValue("Id DESC","idProject = {idProject}") + 1

Chapter 19: Appendix C: Syntaxes for assigning colors

Colors can be typed as readable text, such as "blue", but there are numerous other ways to specify a color, as well as a degree of transparency. This appendix will list these different ways.

Let's start by looking at some examples of color notations used in Proteus:

Type	Example	Comments
HTML hex color	#ff0000	Each byte (00-ff) following the #-sign represents the R (Red), G (Green), B (Blue) value of the color. #ff0000 means "All Red and no green or blue". This type of color will be shown when you choose a color from this color picker: 
C++ hex color	0xFFC0FF	Similar to above but with other prefix.
Visual Basic hex color	&HFFC0FF	Similar to above but with other prefix.
Named colors	Gray or gray	For the full list of named colors, open color picker in e.g Gantt Bar editor (in the Web Colors tab). 
2 color transition	red-green:50	Format: Color1-Color2:TransitionPercent
Complementary color	!red	Results in cyan, which is the complimentary color of red
Readable B/W color	?yellow	Results in either black or white, and in the example at left black would be returned, since white would not be readable with yellow.

Setting transparency:

Any of the color assignments (except the "?color" syntax) can have an additional parameter that controls the opacity (the higher the number, the less transparent it will be). Simply add "**,N%**" where N is an integer between 0 and 100.

Examples:

Red,**20%**	This results in a vaguely visible red
green-red:30,**60%**	Result is a brownish color with some transparency.
!white,**50%**	Result is grayish (complementary of white is black)

Note: It is possible to drop the "%" sign and use a number from 0 to 255 instead, but the % syntax is more understandable and therefore preferable.

Example: **Red,255** is a "completely red" color (no transparency).

Chapter 20: Appendix C2: Syntax for controlling Style in Grid Cells

*Proteus has its own syntax for defining a "Style", i.e a combination of font and color properties. This is done using **one** string with a number of sequenced parameters. This type of formatting may be used to control cell appearance in a grid, but also in Custom Exports.*

The format:

FontStyle§ForeColor§BackColor§BackColor2§FontSize§FontName

Explanations for each of the parameter (all of them are optional):

Parameter function	Description
FontStyle	One or more of the following items (separated with ","): **Regular, Bold, Italic, Underline, Strikeout**
ForeColor	The color of the text in cell. See syntax for coloring here.
BackColor	The background color in the cell/shape.
BackColor2	The gradient background color (if omitted only the background color is used as a solid background).
FontSize	The size of the font.
FontName	The family name of the font.

Examples:

Parameter function	Description
§red	Red text color
Bold,Italic§§§12	Bold italic green font in size 12
§§Blue§DodgerBlue	A gradient background from Blue to DodgerBlue
§Green§Red	Green text on red background

Example of usage in a calculated column:

Let's say you want to show a **blue** text in a column **MyColumn** if **FieldX > 100**, and red background if **FieldY < 50**. The font should always be bold and have fontsize 10:

Note: This could be accomplished using advanced conditional formatting (see Proteus Documentation), but in some cases you may not want to leave it up to the Grid's Layout to define appearance of cells, but instead have them controlled from business logic defined in Configuration. The latter could then be achieved using a calculated column "**_CellFormat_MyColumn**" with this calculation:

```
'Bold§' + Iif(FieldX > 100, 'Blue','Black') + '§' + Iif(FieldY < 50,
'Red','Transparent') + '§§10'
```

The above calculated column will then return any of the four results:

Bold§Blue§Red§§10
Bold§BlackRed§§10
Bold§Blue§Transparent§§10
Bold§Black§Transparent§§10

When Proteus updates appearance for cells in a grid, it first checks if there exists a column in the Grid's DataSet with the same name as the cell's column (in our case it would look for "**_CellFormat_MyColumn**", and if it finds it, the resulting style code will be used to set the appearance (style) of the cell.

Chapter 21: Appendix D: Internally generated fields

There are a number of fields in a DataSet that are added to the ones coming from the DataSource (normally through an SQL statement). Some of these are generated by Proteus internally, and are described in this appendix.

Internally generated columns:

Column Name	Data Type	Description
_Counter	System.Int32	Always 1. Usable in pivoting for getting count.
_ParentStart	System.DateTime	The Parent's Start Date
_ParentFinish	System.DateTime	The Parent's Finish Date
_ParentID	System.String	The Parent's Unique ID
_ParentDescription	System.String	The Parent's Description
_ProgressDate	System.DateTime	The date corresponding to tip of progressbar.
Original<StartColumn>	System.DateTime	The original start date, used for restore
Original<FinishColumn>	System.DateTime	The original finish date, used for restore
_ExpectedProgress	System.Double	The expected progress acc. To TimeNow
_OnSchedule_Start	System.DateTime	The start date, if it had been on schedule
_OnSchedule_Finish	System.DateTime	The finish date, if it had been on schedule
_OnSchedule_Offset	System.Int32	The number of days to move in order to be on schedule.
_IsInWorkSchedule	System.Boolean	True if within the WorkSchedule range
_ChildCount(N)	System.Int32	The number of child-rows. If more than one set of child-data exists, the second set will be called _ChildCount2, etc.
_HasShadow(N)	System.Boolean	True if the row has a related shadow row. If more than one set of child-data exists, the second set will be called _HasShadow2, etc.
_IsOrphan(N)	System.Boolean	True if parent is missing. If more than one child DataSet is defined, the second set will be called _IsOrphan2, etc.
_Duration	System.Int32	The number of days between start and finish dates.
_WorkDays	System.Int32	The number of working days between start and finish dates, according to the calendar that the object is assigned to.

Inherited Columns:

Inherited columns will be prefixed: **_Parent_**<FieldName>

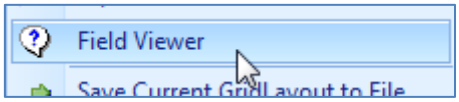
Snapshot Shadow Columns:

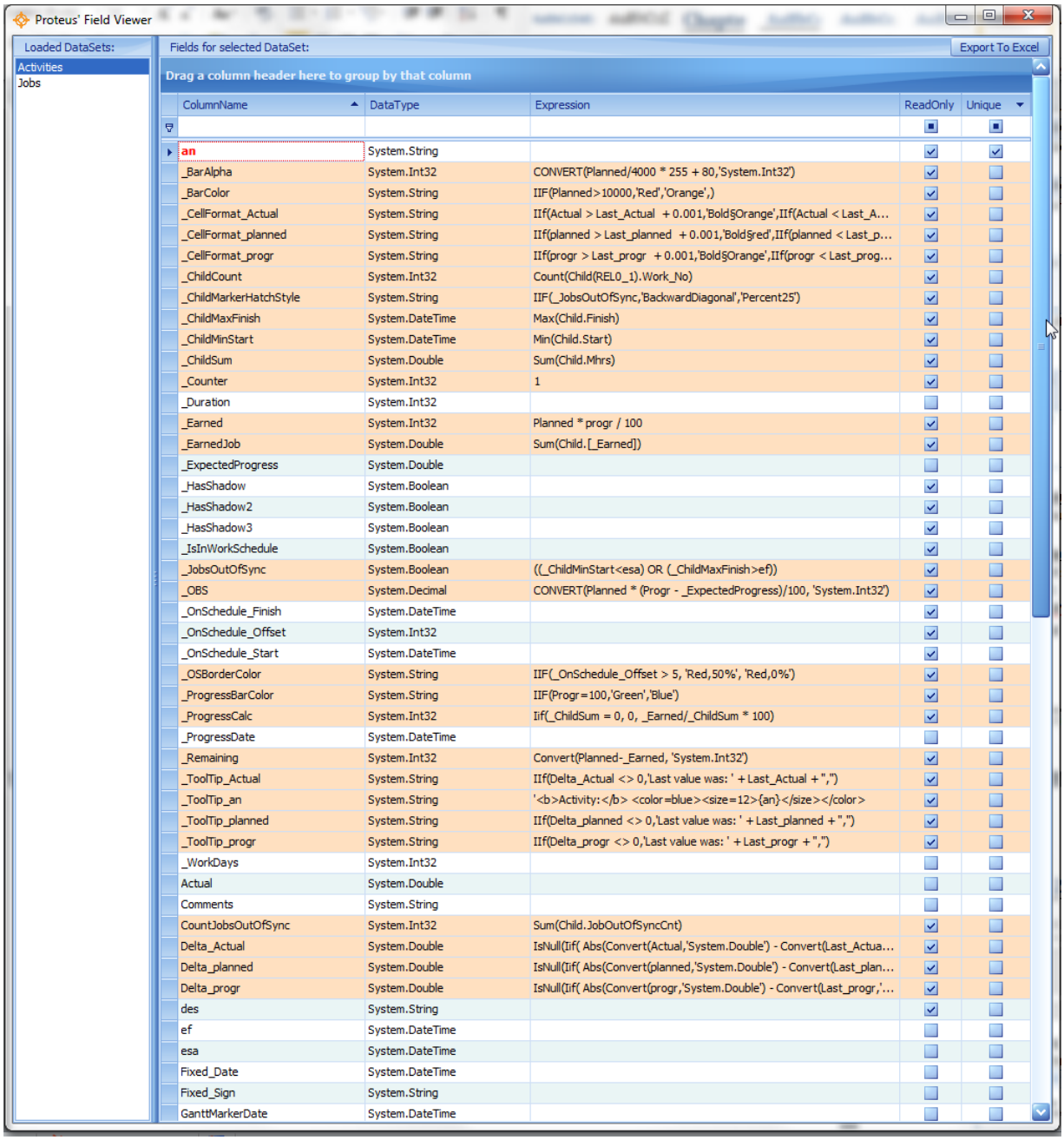
For snapshot shadow columns there are 3 fields generated for each item in "Fieldlist Snapshot Compare" list:

Delta_<FieldName>, **_CellFormat_**<FieldName>, **_ToolTip_**<FieldName>

And one field for each item in the "Fieldlist Select" list, resulting in field: "**Last_**<FieldName>"

There is one tool in Proteus (at runtime) that may be mentioned in this context, that provides a complete list of all DataSet fields, including useful information on their details (such as if it is calculated or not, and its data type).
This list can be exported to Excel (button in top right) and can then act as a useful part of the technical documentation for the Proteus Workspace:



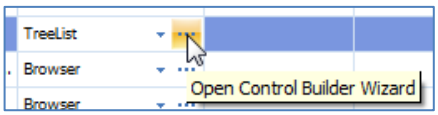


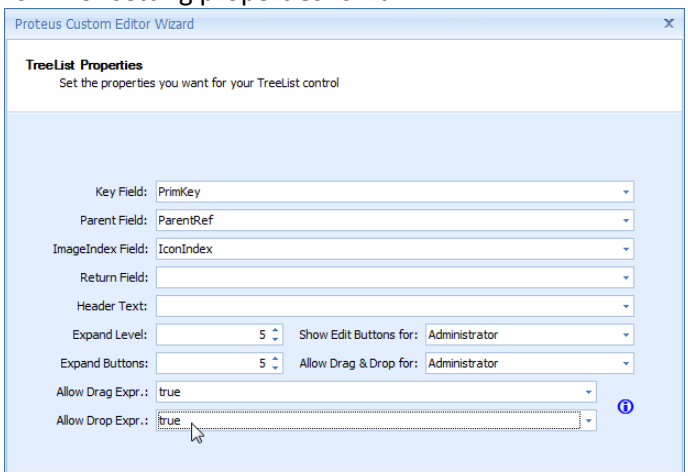
ColumnName	DataType	Expression	ReadOnly	Unique
an	System.String		☑	☑
_BarAlpha	System.Int32	CONVERT(Planned/4000 * 255 + 80,'System.Int32')	☑	☐
_BarColor	System.String	IIF(Planned>10000,'Red','Orange',)	☑	☐
_CellFormat_Actual	System.String	IIf(Actual > Last_Actual + 0.001,'Bold§Orange',IIf(Actual < Last_A...	☑	☐
_CellFormat_planned	System.String	IIf(planned > Last_planned + 0.001,'Bold§red',IIf(planned < Last_p...	☑	☐
_CellFormat_progr	System.String	IIf(progr > Last_progr + 0.001,'Bold§Orange',IIf(progr < Last_prog...	☑	☐
_ChildCount	System.Int32	Count(Child(REL0_1).Work_No)	☑	☐
_ChildMarkerHatchStyle	System.String	IIF(_JobsOutOfSync,'BackwardDiagonal','Percent25')	☑	☐
_ChildMaxFinish	System.DateTime	Max(Child.Finish)	☑	☐
_ChildMinStart	System.DateTime	Min(Child.Start)	☑	☐
_ChildSum	System.Double	Sum(Child.Mhrs)	☑	☐
_Counter	System.Int32	1	☑	☐
_Duration	System.Int32		☐	☐
_Earned	System.Int32	Planned * progr / 100	☑	☐
_EarnedJob	System.Double	Sum(Child.[_Earned])	☑	☐
_ExpectedProgress	System.Double		☐	☐
_HasShadow	System.Boolean		☑	☐
_HasShadow2	System.Boolean		☑	☐
_HasShadow3	System.Boolean		☑	☐
_IsInWorkSchedule	System.Boolean		☑	☐
_JobsOutOfSync	System.Boolean	((_ChildMinStart<esa) OR (_ChildMaxFinish>ef))	☑	☐
_OBS	System.Decimal	CONVERT(Planned * (Progr - _ExpectedProgress)/100, 'System.Int32')	☑	☐
_OnSchedule_Finish	System.DateTime		☑	☐
_OnSchedule_Offset	System.Int32		☑	☐
_OnSchedule_Start	System.DateTime		☑	☐
_OSBorderColor	System.String	IIF(_OnSchedule_Offset > 5, 'Red,50%', 'Red,0%')	☑	☐
_ProgressBarColor	System.String	IIF(Progr=100,'Green','Blue')	☑	☐
_ProgressCalc	System.Int32	Iif(_ChildSum = 0, 0, _Earned/_ChildSum * 100)	☑	☐
_ProgressDate	System.DateTime		☐	☐
_Remaining	System.Int32	Convert(Planned-_Earned, 'System.Int32')	☑	☐
_ToolTip_Actual	System.String	IIf(Delta_Actual <> 0,'Last value was: ' + Last_Actual + ",")	☑	☐
_ToolTip_an	System.String	'Activity: <color=blue><size=12>{an}</size></color>	☑	☐
_ToolTip_planned	System.String	IIf(Delta_planned <> 0,'Last value was: ' + Last_planned + ",")	☑	☐
_ToolTip_progr	System.String	IIf(Delta_progr <> 0,'Last value was: ' + Last_progr + ",")	☑	☐
_WorkDays	System.Int32		☐	☐
Actual	System.Double		☐	☐
Comments	System.String		☐	☐
CountJobsOutOfSync	System.Int32	Sum(Child.JobOutOfSyncCnt)	☑	☐
Delta_Actual	System.Double	IsNull(Iif(Abs(Convert(Actual,'System.Double') - Convert(Last_Actua...	☑	☐
Delta_planned	System.Double	IsNull(Iif(Abs(Convert(planned,'System.Double') - Convert(Last_plan...	☑	☐
Delta_progr	System.Double	IsNull(Iif(Abs(Convert(progr,'System.Double') - Convert(Last_progr,'...	☑	☐
des	System.String		☑	☐
ef	System.DateTime		☐	☐
esa	System.DateTime		☐	☐
Fixed_Date	System.DateTime		☐	☐
Fixed_Sign	System.String		☐	☐
GanttMarkerDate	System.DateTime		☐	☐

Chapter 22: Appendix E: Working with a TreeList control

The TreeList control allows data to be presented in a hierarchical tree format. It can be defined as part of a DataLayout, and offers the option of editing the hierarchical structure of its nodes. In some situations it may be necessary to have rules that control what is allowed when dragging and dropping nodes in the tree.

When adding a TreeList in a DataLayout, the wizard brings up a form for setting properties for it:





Setting up a TreeList

Property	Description
Key Field	The field with unique values for each node.
Parent Field	The field referring to a Key Field value (parent).
ImageIndex Field	(optional) The field that holds the index to the icon to use (*).
Return Field	(optional) The field to be returned (as a collection of selected fieldvalues).
Header Text	(optional) The text to show in header. Supports markup of text.
Expand Level	The number of levels to expand to initially.
Expand Buttons	The number of "expand-level-buttons" to show.
Show Edit Buttons for	The Groups to show the Edit Buttons for.
Allow Drag & Drop for	The Groups to allow drag and drop for. More info below.
Allow Drag Expr.	The Expression that controls if a node can be dragged or not. More info
Allow Drop Expr.	The Expression that controls if a node can be dropped or not. More info

(*) the images are loaded from the folder "IconImages" and their "index" is defined by filename (info here).

Using a TreeList in a Custom Dialog

In case a TreeList control is used in a Dialog (and only in this case), you need to define the "Return Field" (see list above). When such a field is defined, the TreeList will be showing checkboxes at the left end of each node, allowing it to be selected/deselected.

You will want to use the items that are "checked off" in some meaningful way, so an Expression is required to help you in this process.
Here is an example that converts the collection of selected nodes and returns a comma separated string of items:

```
String.Join(",",General.ToStringArray(dlg#ProjectListEPS#SelectedProjects.ToArray()))
```

Set rules for drag & drop of nodes in TreeList

Allow Drag Expr. and **Allow Drop Expr.** Expressions control if node can be dragged from or dropped upon, respectively. These "Markers" are available in the expression:

Marker	Description
DragNode	Node Object representing the node being dragged
DropNode	Node Object representing the node being dropped upon
InsertDirection	String: "Before" , "After" , "None"
DragNodePath	String: A ';'-separated list of node names of drag node path
DropNodePath	String: A ';'-separated list of node names of dragged node path
DragNodeChildList	String: A ','-separated list of field values of last defined field

Note that since the **DragNode** and **DropNode** represent objects, they will have properties that you can access. Some useful ones are:

Property	Description
Level	A zero based value indicating tree hierarchy level.
GetValue(N)	The value in tree column N. Note that N does NOT indicate the visual order, but the order in which it is defined.
ParentNode	The Node Object representing the parent node of the current one. You may refer to a grandparent node of the dragged node like this: DragNode.ParentNode.ParentNode

A few examples to show some possibilities:

Example description	Expression
Only allow drop on same hierarchical level	`(DropNode.Level == DragNode.Level) && (InsertDirection != "None")`
Only allow dragging items that do not have "Completed" in 3rd column of grid. Set the CanDrag Expression as shown.	`DragNode.GetValue(2) != "Completed"`
Setting the CanDrop Expression as shown will have the following effect: Only allow dropping a node above or below (not as child) a node that is on same level as it was dragged from. Two exceptions to this rule: 1. A node is dragged from Level 1 and dropped as a child on a root level node on a different branch of tree. 2. A node is dragged from Level 1 and dropped above or below another Level 1 node, on a different branch of tree.	`General.Iif((DragNode.Level==2) && (((DropNode.Level == 2) && (InsertDirection != "None") && (DragNode.ParentNode.Id == DropNode.ParentNode.Id)) || ((DropNode.Level == 0) && (DragNode.ParentNode.ParentNode.Id != DropNode.Id) && (InsertDirection == "None"))), true, General.Iif((DragNode.Level==1) && (((DropNode.Level == 1) && (InsertDirection != "None")) || ((DropNode.Level == 0) && (InsertDirection == "None"))) , true, false))`

Chapter 23: Appendix F: Markup syntax to format text

It is sometimes quite useful to emphasize information in tooltips (for cells and column headers). This is possible to achieve by using the limited, but useful, set of markup commands to insert in the text you wish to show.

Tag	End Tag	Description
** **	-	Inserts a single line break.
<color=*value***>** Examples: <color=red> <color=0,255,0> <color=#0000FF>	**</color>**	Specifies the text color.
<size=*value***>** Examples: <size=10> <size=+4> <size=-4>	**</size>**	Specifies the font size.
****	****	Defines bold text.
****	****	Defines italic text.
<u>	**</u>**	Defines underlined text.

Size = 14
Bold *Italic* Underline
Size = 11
Sample Text

The above result is produced using the markup below:

```
<size=14>Size = 14<br>
<b>Bold</b> <i>Italic</i> <u>Underline</u><br>
<size=11>Size = 11<br>
<color=255, 0, 0>Sample Text</color>
</size>
```

These are the contexts where you can use these types of markup:

1. ToolTips in Grid (GridGantt and Completion Matrix) / TreeList.
2. TreeList Header Text.
3. ToolTip Labels in a DataLayout (when changing the design of a DataLayout in runtime).
4. Shape Text in a Custom Export will render the markup in shapes in Excel and PowerPoint.

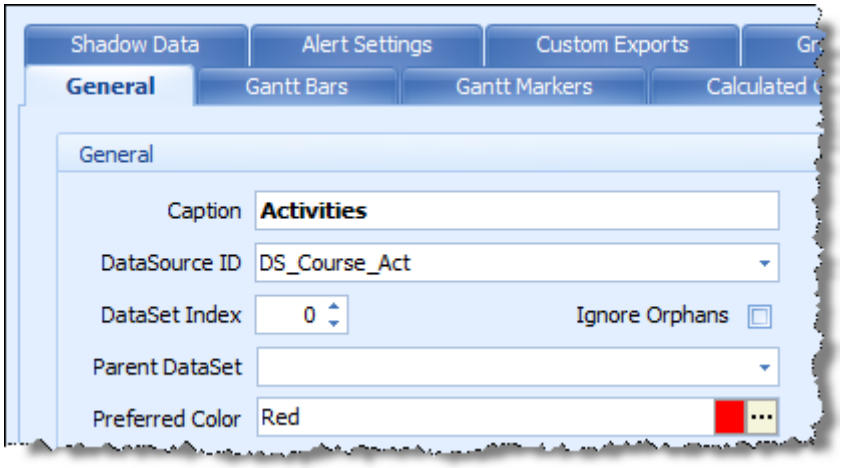
Chapter 24: Appendix G: Exporting data to Safran Planner

Proteus can export its hierarchical Grid and Gantt Chart to Safran Planner format. From Proteus this is done by right clicking in a GridGantt and selecting "Export to Safran Planner".

There is a way to control the logo shown in Safran Planner printouts:
If you have provided these files, they will appear in header as left and right logo:
.\ReportImages\SPPLogo_left.bmp
.\ReportImages\SPPLogo_right.bmp

NOTE: Safran Planner only supports .bmp format.

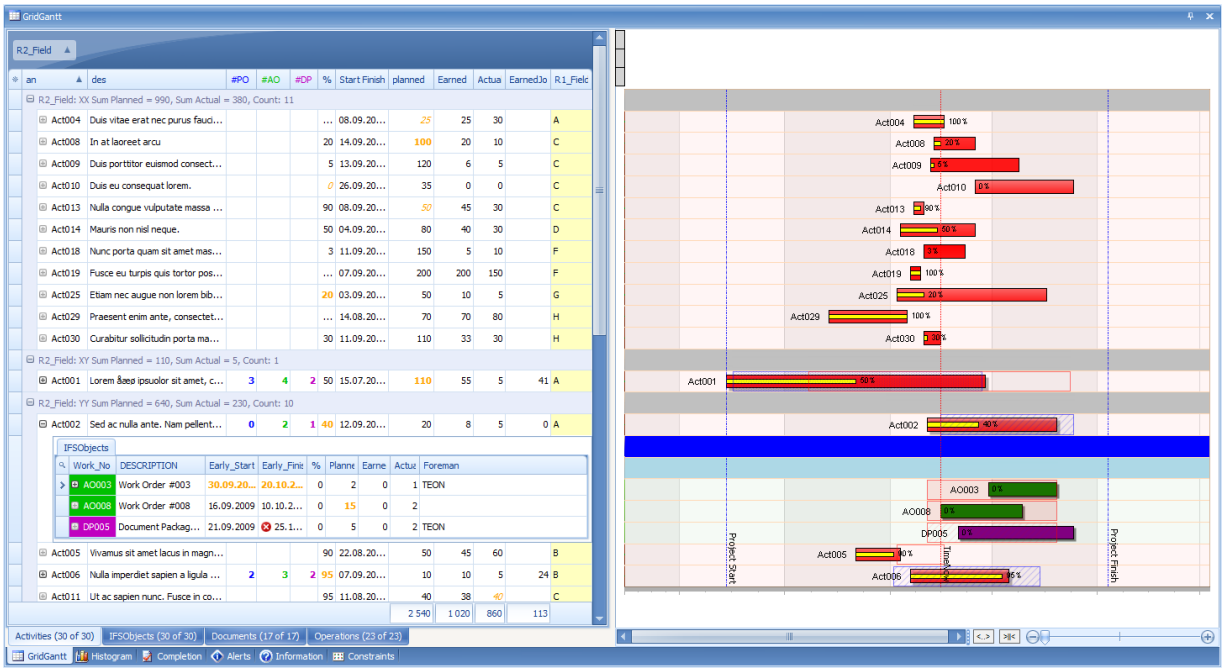
The colors of the bars will be the same as the "Preferred Color" for the DataSet that "owns" the bars:



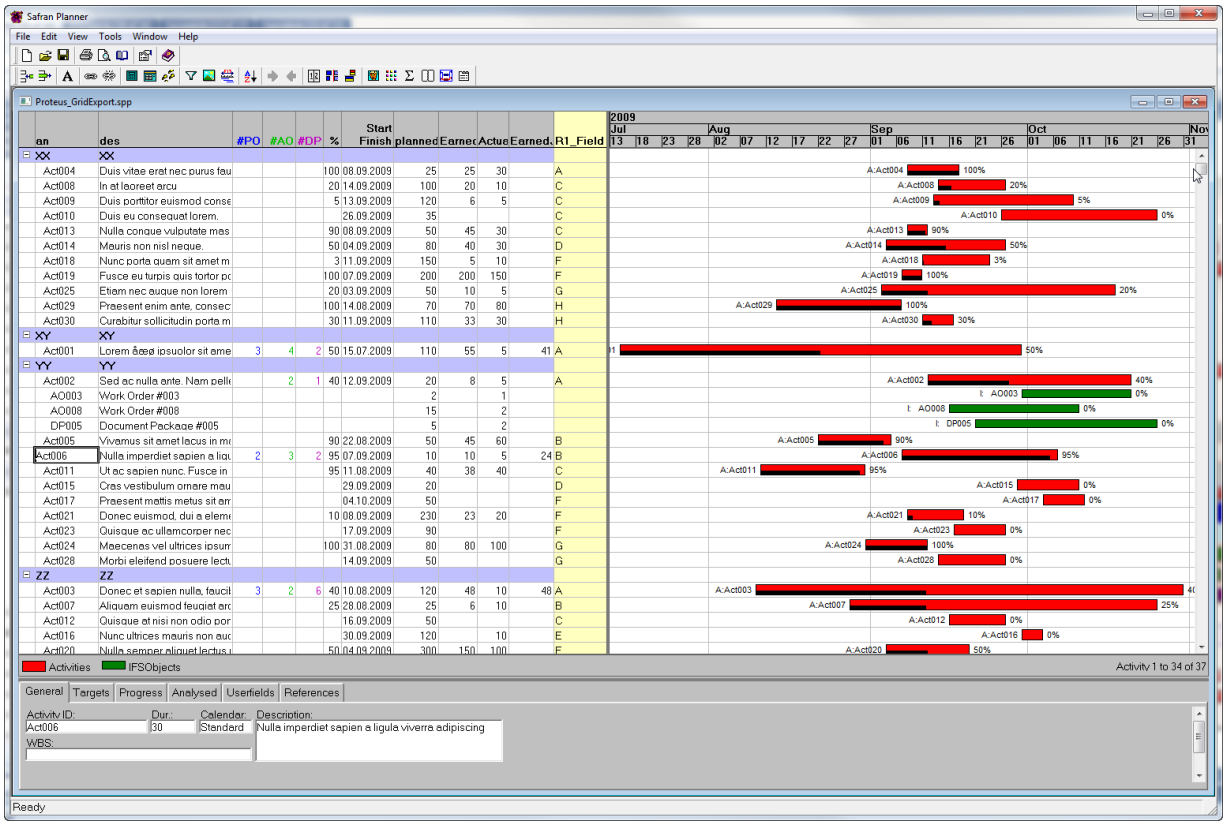
The next page shows a screenshot from Proteus and the result after export to Safran Planner. There are a few things to comment on there:

1. The background and text colors are "transferred" to the result, but individual cell variations are not. You will notice in example on next page that some cells are orange in Proteus, while black in Safran Planner.
2. If you look at a child DataSet (i.e the DataSet that is not topmost), if the text (caption) given in column headers match text in column headers at toplevel, then this field's values will show in the same column. This can be quite useful when "reusing columns" for different levels of data.
3. The "expanded/collapsed status" of a row will be reflected in the exported result. If you wish to expand all children (down to some hierarchical level), you can right click in grid and choose the Expand option there.
4. The color of progress bar is not "transferred" to Safran Planner result, nor are, as mentioned previously, the individual bar-colors.
5. Setting a background/foreground color for a column in Proteus, transfers with same colors to result in Safran Planner.

Below is an example of export to Safran Planner:



Screenshot of Proteus



Screenshot from result in Safran Planner.

Chapter 25: Appendix H: Working with Project Folders

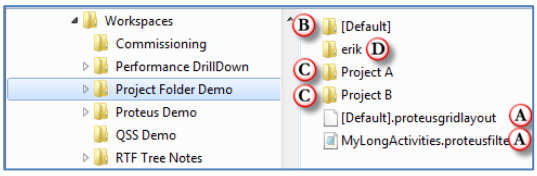
*There is a feature in Proteus that allows various users to share a common set of layouts, filters, reports, etc (here called Customization Files, or "C-Files" for short). You can set up an expression that determines which such **set** of user configurations (the name of a folder) to use.*

Let's use a specific example case to get started…

Let's assume we want to achieve the following functionality:
The user should select which project whey work for in an initial custom dialog.
The dropdown should have 2 project names – "**Project A**" and "**Project B**", and each should have their separate sets of reports, filters, layouts, etc (C-Files).
However, there should be a set of such items that are considered "common defaults", i.e settings that initially are common to both, but can be overridden.
An example to illustrate this could be to have an initial filter called "MyLongActivities" that filters out activities having more than 44 day durations. **Project A** decides they want to overwrite this filter with one having 50 days instead, and **Project B** wants 60 days.

This is how one would achieve the above "mission":

1. Cut (Ctrl-X) all your existing C-files (A) in your particular Workspace-folder, and create a new folder under it called "**[Default]**" (B), where you paste them in. In the above example you would, among other files, move your filter "MyLongActivities" (that filters out activities with durations over 44 days).



Folder structure for Project Folders

2. Create two folders called "**Project A**" and "**Project B**" (C).
   Note: The other folder(s), (D), have been automatically created by Proteus to store Private C-Files. They should also be moved into your new [Default] folder, if you want to keep private settings for each user.
3. Set up a custom dialog with a dropdown having the 2 choices (Project A and Project B):
   Create a dialog called "MyProjectSelectorDialog" having a combobox, called "MyProject" with the two items in it.
4. Set up an Expression called MyProjectFolderName and have its expression refer to your custom dialog's combobox: dlg#MyProjectSelectorDialog#MyProject.
   Note: If, for whatever reason, your expression results in a folder name that does not exist, the [Default] folder settings will be used.
5. Type "MyProjectFolderName" in the "Project Folder" box in General Workspace properties.
6. Start Proteus, and select "Project A" in dropdown – double-click on the filter "MyLongActivities". Modify the filter setting from 44 to 50 and right click on "MyLongActivities" and choose "Save Item" to overwrite its settings. In this way you have made a custom version of a "common" filter. A similar modification could be made for Project B.

One improvement to the above example would be to add a choice in the combobox called "[Default]", that should perhaps be shown only to members of an "Admin" Group (In which case the source to the combobox would need to be an expression instead of a fixed string).
This addition would allow "Admin" users to define the list of default layouts, reports, filters, etc that should be available to all underlying projects.

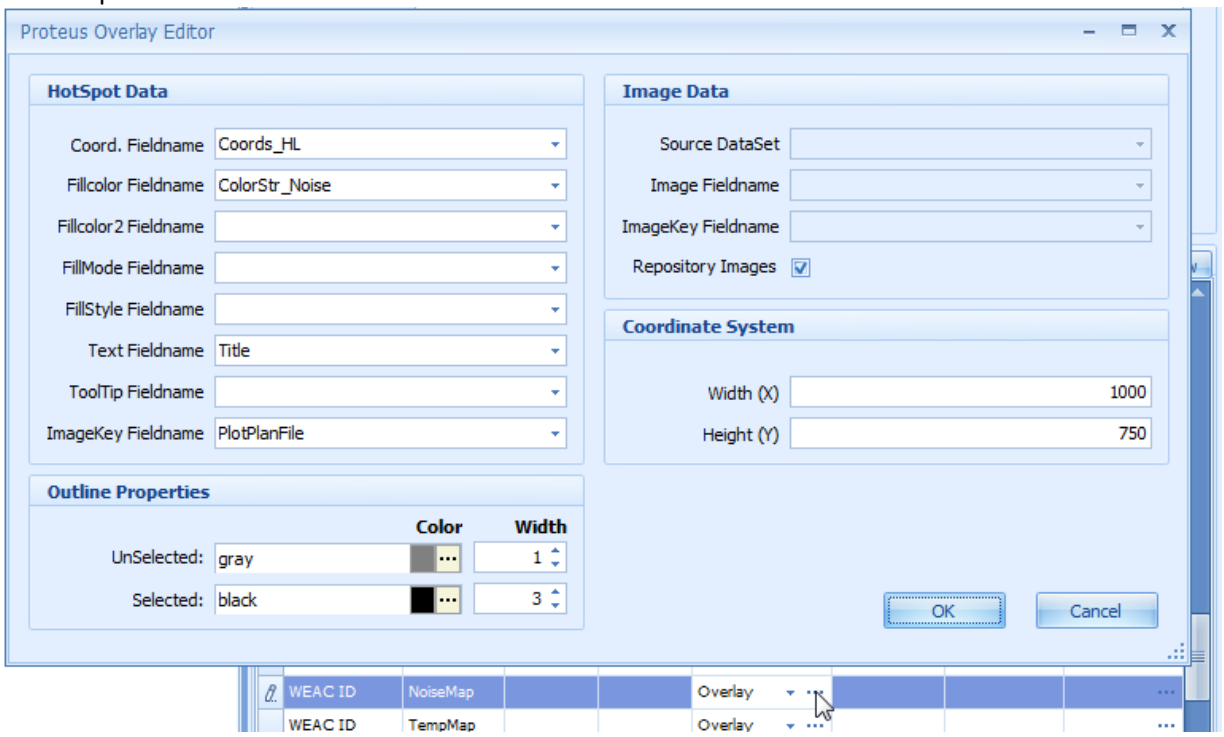
Chapter 26: Appendix I: Working with the Overlay Control

A special DataLayout control, the Overlay Control, has been developed that allows defining "hotspots" on a bitmap(s). Each such hotspot can have a color, transparency, optional gradient/hatch color, as well as being editable – nodes can be added/removed from the polygon to define its geometry.

The Overlay control was developed to allow for two specific things:

1. To make areas of a bitmap "clickable", so that related information for the clicked area can be seen.
2. The area's visual properties should be possible to control from properties in the data, offering a feedback on "*status*" for a particular area.

The ways such a component can be used are quite varied, and at the end of the chapter a couple of screenshots are shown of existing examples.
We will here give a brief screenshot of how the first of these examples (the Color Map for Noise) can be set up:

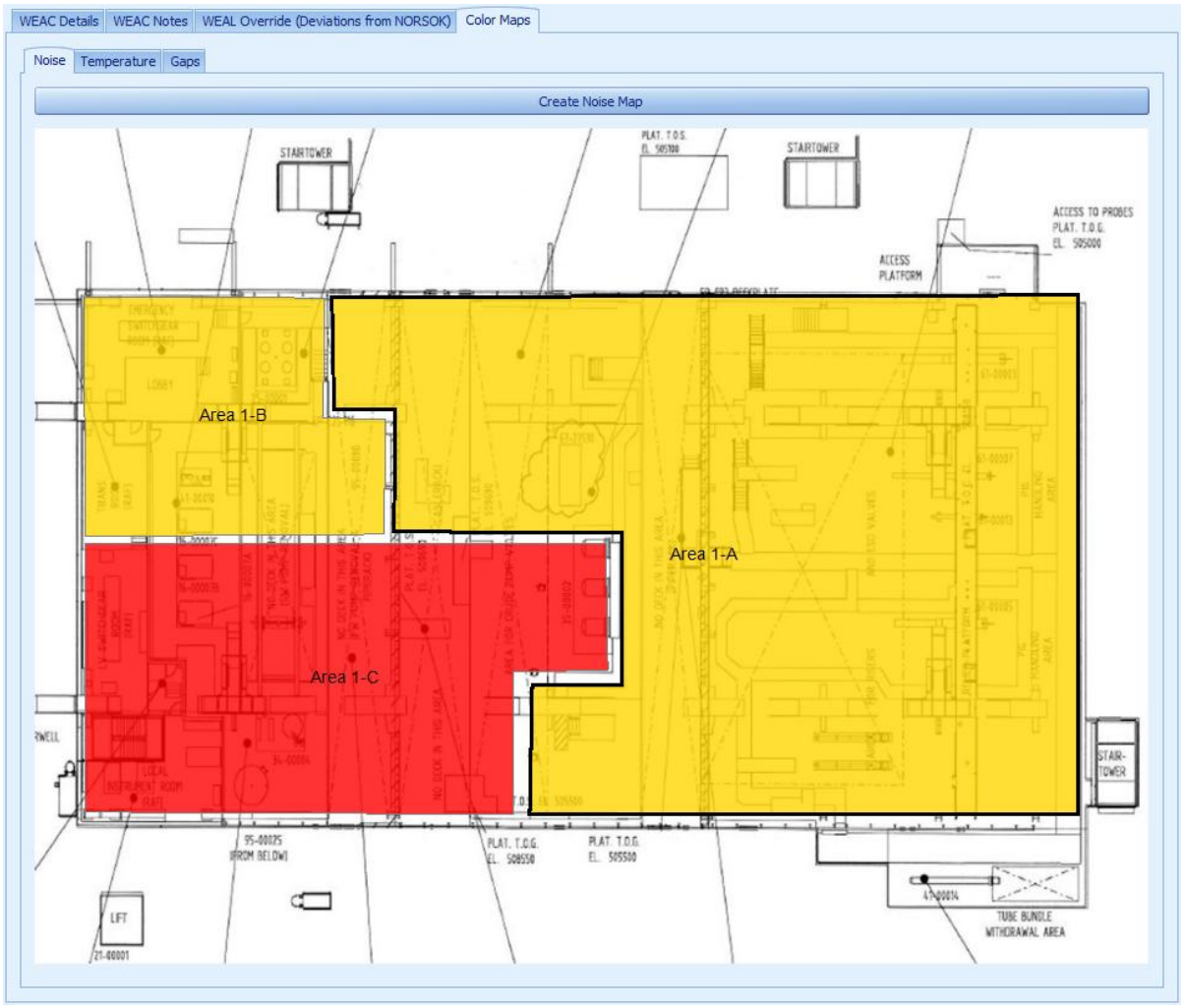


Explanations for the field assigned to an Overlay Control:

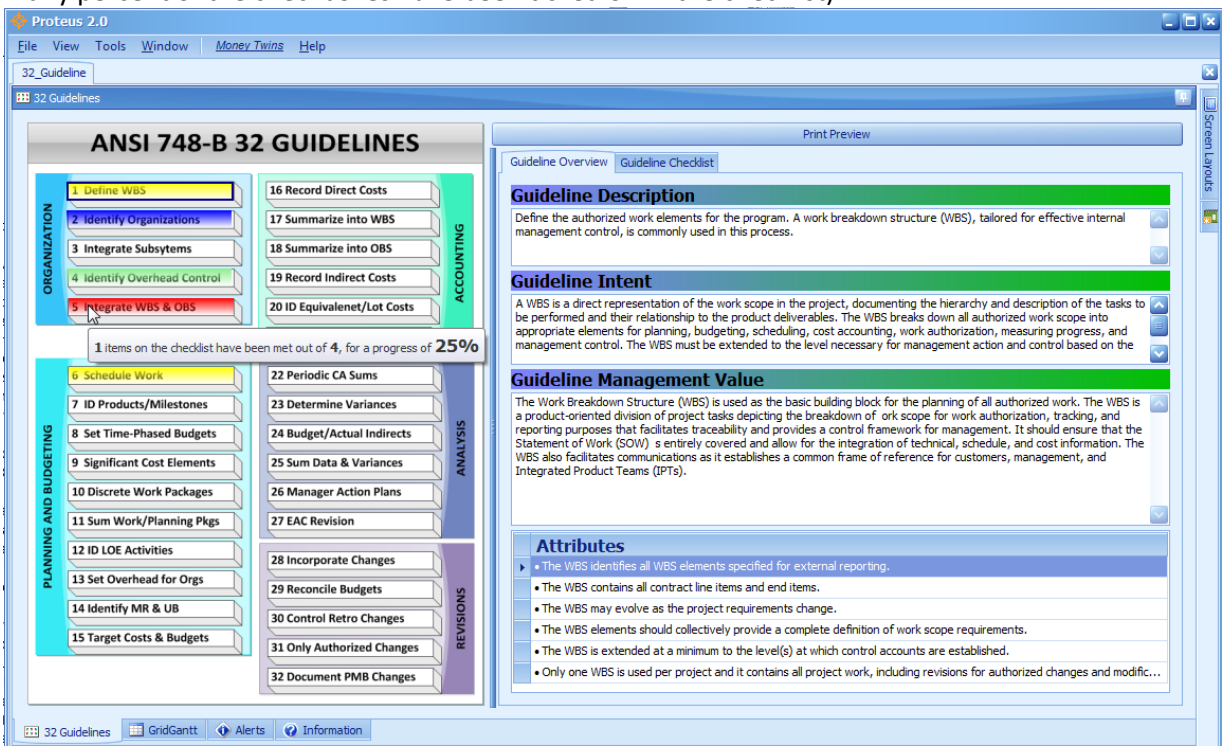
Control in Editor	Description
HotSpot Data	
Coord. Fieldname	Assign which field holds polygon coordinates. Format defined here.
Fillcolor Fieldname	The field that contains information about main shape color.
Fillcolor2 Fieldname	The field that contains info about gradient/hatch color
FillMode Fieldname	The field that contains info about FillMode. Valid values are: **Solid, Gradient, Hatch**
FillStyle Fieldname	This controls properties of the Gradient and Hatch brushes. Typical values for Gradient (full list here):

	LeftToRight, RightToLeft, TopToBottom, BottomToTop Typical values for Hatch (full list here): **BackwardDiagonal, ForwardDiagonal**
Text Fieldname	The field that contains the text to show in shape.
ToolTip Fieldname	The field that contains the tooltip to show for shape.
ImageKey Fieldname	The field that names the image key to use as background image. If "**Repository Images**" is checked, then the images must be placed in the Repository subfolder called "**GUIImages**". Example of calculated field resulting in the ImageKey value: `'{{SelectedInstallation}}' + '_' + IsNull(Slide_HL,'Blank') + '.jpg'`
Image Data	
Source DataSet	(optional) The name of the **DataSet** that contains the image data
Image Fieldname	(optional) The name of the field that contains the actual image data (blob field)
ImageKey Fieldname	(optional) The name of the field that contains the Image Key (string field)
Repository Images	Check if you wish to use images from file system instead, in which case no "Source DataSet" value needs to be defined. See "**ImageKey Fieldname**" above.
Coordinate System	
Width (X) / Heights (Y)	The range of the coordinate system. The polygons defined supplied in "Coord. Fieldname", will be scaled according to this.
Outline Properties	
Define the properties for Selected and Unselected shape's outlines.	

Here is the result:



Here is another typical example, where the color of each polygon is controlled by the progress (how many percent of the checkboxes have been ticked off in the checklist):


Proteus 2.0
File View Tools Window Money Twins Help
32_Guideline
32 Guidelines
ANSI 748-B 32 GUIDELINES
ORGANIZATION
1 Define WBS
2 Identify Organizations
3 Integrate Subsytems
4 Identify Overhead Control
5 Integrate WBS & OBS
ACCOUNTING
16 Record Direct Costs
17 Summarize into WBS
18 Summarize into OBS
19 Record Indirect Costs
20 ID Equivalenet/Lot Costs
1 items on the checklist have been met out of 4, for a progress of 25%
PLANNING AND BUDGETING
6 Schedule Work
7 ID Products/Milestones
8 Set Time-Phased Budgets
9 Significant Cost Elements
10 Discrete Work Packages
11 Sum Work/Planning Pkgs
12 ID LOE Activities
13 Set Overhead for Orgs
14 Identify MR & UB
15 Target Costs & Budgets
ANALYSIS
22 Periodic CA Sums
23 Determine Variances
24 Budget/Actual Indirects
25 Sum Data & Variances
26 Manager Action Plans
27 EAC Revision
REVISIONS
28 Incorporate Changes
29 Reconcile Budgets
30 Control Retro Changes
31 Only Authorized Changes
32 Document PMB Changes
Print Preview
Guideline Overview
Guideline Checklist
Guideline Description
Define the authorized work elements for the program. A work breakdown structure (WBS), tailored for effective internal management control, is commonly used in this process.
Guideline Intent
A WBS is a direct representation of the work scope in the project, documenting the hierarchy and description of the tasks to be performed and their relationship to the product deliverables. The WBS breaks down all authorized work scope into appropriate elements for planning, budgeting, scheduling, cost accounting, work authorization, measuring progress, and management control. The WBS must be extended to the level necessary for management action and control based on the
Guideline Management Value
The Work Breakdown Structure (WBS) is used as the basic building block for the planning of all authorized work. The WBS is a product-oriented division of project tasks depicting the breakdown of ork scope for work authorization, tracking, and reporting purposes that facilitates traceability and provides a control framework for management. It should ensure that the Statement of Work (SOW) s entirely covered and allow for the integration of technical, schedule, and cost information. The WBS also facilitates communications as it establishes a common frame of reference for customers, management, and Integrated Product Teams (IPTs).
Attributes
• The WBS identifies all WBS elements specified for external reporting.
• The WBS contains all contract line items and end items.
• The WBS may evolve as the project requirements change.
• The WBS elements should collectively provide a complete definition of work scope requirements.
• The WBS is extended at a minimum to the level(s) at which control accounts are established.
• Only one WBS is used per project and it contains all project work, including revisions for authorized changes and modific…
32 Guidelines
GridGantt
Alerts
Information
Screen Layouts

Chapter 27: Appendix Ib: The Overlay Control's polygon format

An example may best illustrate the format for a polygon (a hotspot area) in the Overlay Control:

Let's say we have a coordinate system with (0,0) in the lower left corner, and (100,100) in the upper left one, and we want do define a rectangle that has its lower left corner in (0,0) and its upper right corner in (100,50):

```
Polygon: {[0.0,0.0], [0.0,50.0], [100.0,50.0], [100.0,0.0]}
```

Similarly, any kind of shape can be defined using an arbitrarily long list of coordinates.

Note that the World Coordinate system must be defined for a DataLayout Control (Width (X) and Height (Y)), and it gives geometric meaning to the actual coordinate numbers. In this case they would be set to 100 and 100 respectively.

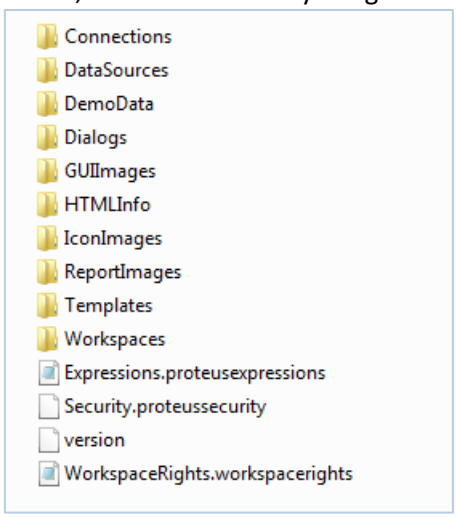
Interactive editing of polygons:
In Proteus an Overlay Control can be right clicked and set in "Design Mode", where coordinates can be added and removed, and points moved. The resulting polygon will be written back to the bound field assigned to contain coordinates for the hotspots.

Chapter 28: Appendix J: Folder Structure and Security

A Repository consists of various xml files that are organized according to a particular folder structure. Some of these folders are automatically generated, while others are optional and must be created manually. The security aspects of these folders are also discussed in this appendix.

Folder structure:

Some subfolders in the repository are created automatically by Configurator, and some are optional – in which case they can be manually created – and populated with files. They are described in table below, and shown visually in figure at right.



Repository's files and subfolders.

Notes on security:

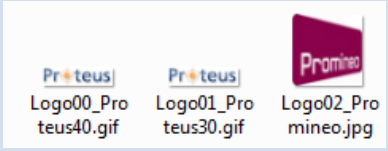
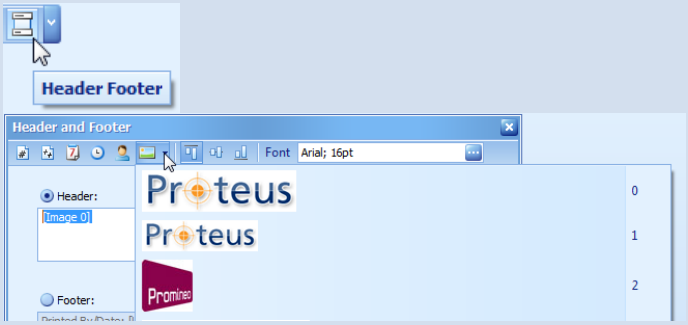
All Proteus users must have read access to the repository folder and most of its subfolders. In order to save layouts, filters and other personal settings, each user will need to have write access to a personal folder under Workspaces/[Name of Workspace]/[Username]. The username is either the windows username (without the domain name) or the Proteus username, as set up on the actual Workspace.

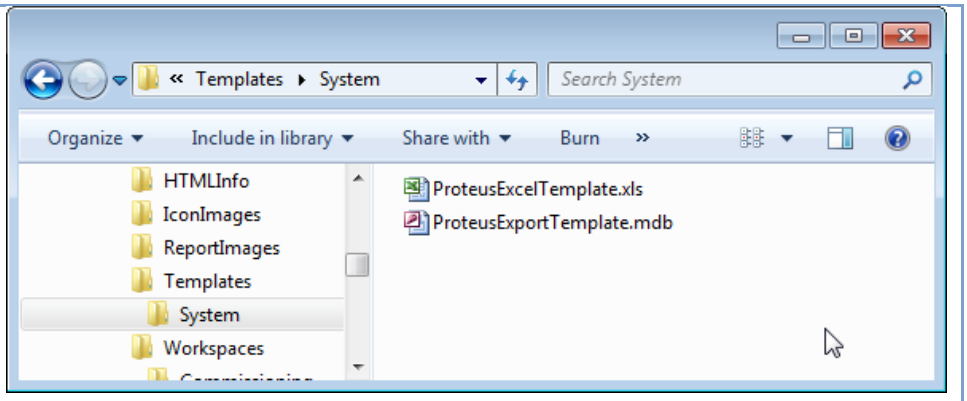
In order to use the Proteus Configurator, the user must have read/write access to the repository folder and all subfolders (except to the personal folders used for each user's private settings).
If you require a simpler level of access control, you can choose to use the built-in access layer of the Proteus Repository. You can define Groups, Users and individual read/write access to each Workspace from the Configurator.
It is important to point out, however, that this feature will *not provide a rigorous protection from any unauthorized access to the repository*, but is mainly intended to prevent accidential changes from being done using the Proteus Configurator. A practical solution is to give all users read access to the repository, but read/write access to the Workspace folder and -subfolders. In this way, you secure the shared configuration in the repository, and simplify file-access administration at the cost of exposing the workspace configurations. Please note that there is an exception to this "practical solution", in the case "Log Usage" is activated, in which case all users must have write access to root folder.

In the folder screenshot above, you see two files that have special significance for access rights within Proteus and Configurator: "**Security.proteussecurity**" and "**WorkspaceRights.workspacerights**".
You may consider using NTFS security for the files to stop unauthorized changes to their content.

Subfolders in a Repository:

As mentioned above, some subfolders are created automatically, and some optional ones can be created manually. Here is a list of the auto-generated, as well as the optional subfolders:

Subfolders	
Connections	Auto-generated to hold DataConnection files.
DataSources	Auto-generated to hold DataSource files
Dialogs	Auto-generated to hold Custom Dialog files.
Workspaces	Auto-generated to hold Workspace settings. Will automatically get one subfolder for each created Workspace.
GUIImages	Manually created to hold images to use in custom Dialogs and DataLayouts (PictureEdit control type).
IconImages	Manually created to hold images for use in these settings: 1. DataLayout's TreeList control (icon images in tree) 2. CaptionImage for DataLayout controls. 3. Custom Grid Column Editor control of type "ImageDropdown". There is an ***important naming convention*** for the image filenames that controls how an icon index (given in DataLayout's ImageIndex property or a IconIndex in a TreeList) can match a corresponding file. Here is an example to make this clearer: MyIconFile_**001**.jpg <- The numbers immediately in front of the file extension will be used as the **icon index**.
ReportImages	Manually created to hold images used for reporting. The naming convention for files in this folder is "**LogoNN_xxx.eee**". Explanation: "xxx" can be any characters. "eee" can be any extension characters. "**NN**" must be 2 digits, starting with **00**. As seen from the dropdown list found in header footer setup, the numbers in right margin match the number NN:  

Templates	 Manually created and should contain a subfolder called "System" that contains the files you wish to use as templates when generating Histograms in Excel, as well as dumping loaded data into MSAccess. Note that you must use these specific filenames (the Excel template can also be with extension .xltm): **ProteusExcelTemplate.xls** **ProteusExportTemplate.mdb**
HTMLInfo	Manually created and should preferably have subfolders containing help info for the various Workspaces. The name of the .html or .doc file containing the help information can be given in General Tab of a Workspace.
"DemoData"	Manually created, and the name can be arbitrary. If you have any data sources you want to be contained within the repository, then it is a good idea to include them (.mdb, .accdb, .xls, .csv files) in a subfolder.

Note that the subfolders in quotes can have an arbitrary name, the others, if manual, must match exact spelling.

Note:
When a new Repository is created, there exists no Users, Groups, or Workspaces. When creating Workspaces, you can define which Groups should be able to see and modify it.
As long as there are no defined users, anyone can open/edit any Workspace in the Repository.

Chapter 29: Appendix K: Formatting syntax for Time Units

The time scale of the Histogram / Gantt Chart can be customized in various ways. The same syntax is used when setting up the Time Unit choices available for Histogram.
This appendix gives an overview of the various ways to format a time entity to a string.

The formatting options are quite extensive, and more info is found in these links:
Standard formatting: http://msdn.microsoft.com/en-us/library/az4se3k1.aspx
Custom formatting: http://msdn.microsoft.com/en-us/library/8kb3ddd4.aspx

Special Proteus specific formatting codes:
We have extended the possibilities shown in the links above, with the following codes:

[Week]	Returns the year and week in this format: 13-51 (2 digits for year and week)
q	Returns a one digit quarter number.
w	Returns a two digit week number (starting with 01 and ending with 52 or 53)
d1	Returns the day name's first letter in lower case.
D1	Returns the day name's first letter in UPPER case.
m1	Returns the month name's first letter in lower case.
M1	Returns the month name's first letter in UPPER CASE.

In Proteus (by right clicking the Gantt's time scale), this list can be seen showing typical examples to choose from (or you can type your own custom formatting code):

Item	Format	Example
Year Long	yyyy	2012
Year Short	yy	12
Quarter	Qq	Q1
Month Text	MMM	Jan
Month Letter	M1	J
Month Number	MM	01
Week Number	ww	08
Day Letter	D1	M
Day Number	dd	01

If you wish to include a letter that you want to avoid converting to an equivalent formatting result, you can prefix the letter with a "\" character. An example will clarify this:
The format "**dd**-**q**" would result in this (for example): **01**-**1**
The format "**d****d**-**q**" would result in this (for example): **dd**-**1**

Copyright Notice

Copyright © 2012 Promineo AS

All rights reserved. No part of this publication may be reproduced in any manner without the prior agreement and written permission of the publisher.

Published by: *Proteus Development Team, Eiganesveien 8, Stavanger, Norway*

Date: *December, 2012*

Feedback: *We always try to improve our product, including its documentation. Please feel encouraged to report any errors or other feedback regarding this document to:*
proteus@promineo.no